Radio Shack®

TRS-80™ Model II BASIC Reference Manual

A Description of the Model II BASIC Programming Language: Definitions, Syntax, Examples and Sample Programs

CUSTOM MANUFACTURED IN THE U.S.A. BY RADIO SHACK A DIVISION OF TANDY CORPORATION

BASIC

TRS-80™ Model II

BASIC Reference Manual

BASIC Version 1.2

Radio Shack®
A DIVISION OF TANDY CORPORATION
One Tandy Center
Fort Worth, Texas 76102

Printed Material Copyright Notice
First Edition — 1979

Model II BASIC
Software Copyright Notice

Printed in the United States of America

10 9 8 7 6 5 4 3 2

Contents

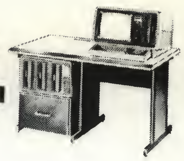
CHAPTERS

APPENDIX

INDEX

Chapter 1

Using Model II BASIC

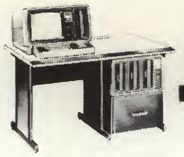


General Information

Model II BASIC is an easy-to-use, extended version of the BASIC programming language. It is designed to run under the TRS-80 Disk Operating System (TRSDOS), and is included on the System diskette.

Model II BASIC executes your programs directly. It does not produce a low-level, machine-language translation. In technical terms, it is an interpreter, not a compiler. This makes it especially powerful for interactive use during program development and debugging.

Model II BASIC offers all the standard features of the language, plus several important additions, including:
- Program line renumbering
- Line editor for easy program corrections and changes
- Ability to execute a TRSDOS command and return to BASIC with program and variables intact
- Direct and sequential access to data in disk files
- Special functions to allow BASIC programs to call machine-language subroutines
- Recovery from operator errors—the System won't stop if you attempt output to a device (such as a Printer or Disk Drive) which is not ready.



Notation

For clarity and brevity, we use some special notation and type styles in this manual.

CAPITALS and punctuation
Indicate material which must be entered exactly as it appears. (The only punctuation symbols not entered are ellipses, explained below.) For example, in the line:

```
PRINT "THE TIME IS  " TIME$
```

every letter and character should be typed exactly as indicated.

lowercase italics
Represent words, letters, characters or values you supply from a set of acceptable values for a particular command. For example, the line:

LIST *line-range*

indicates that you can supply any valid line-range specification after LIST.

. . . (ellipsis)
Indicates that preceding items can be repeated. For example:

INPUT *variable, . . .*

indicates that several variables may be repeated after INPUT.

b̸
This special symbol is used occasionally to indicate a blank space character (ASCII code 32). For example:

```
BASIC b̸ PROG
```

The b̸ indicates that there is a single blank space after BASIC.

[aaaa,bbbb]
Indicates a numeric range with lower limit *aaaa* and upper limit *bbbb*. Both limits are included in the range. For example:

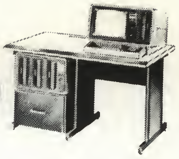
```
[−32768,32767]
```

represents the range of numbers from −32768 to 32767 inclusive. The context will specify whether integers or real numbers are intended.

X'NNNN'
Indicates that NNNN is a hexadecimal number. Numbers used in this manual are in decimal form, unless otherwise noted. For example:

```
X'700A'
```

is a hexadecimal representation of the decimal number 28682.



O'NNNNN'

Indicates that NNNNN is an octal number. Numbers used in this manual are in decimal form, unless otherwise noted. For example:

```
O'17707'
```

is an octal representation of the decimal number 8135.

keyname

Indicates one of the keys, usually a special control key like ENTER . For example:

```
PRINT "THE TIME IS " TIME$       ENTER
```

indicates you should press ENTER after typing in the text.

CTRL keyname

Indicates a control character. To output the character, hold down CTRL and press the specified key. For example:

```
CTRL R
```

Indicates that you should hold down CTRL and press R .

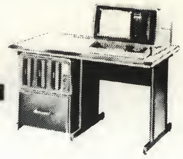
About This Reference Manual

This manual describes the keywords, data types, and other features which are available in Model II BASIC. You'll find plenty of examples and sample programs to help you try out the language. There is also a Glossary in the Appendix.

The manual is organized this way:

Chapter 1. Using Model II BASIC
A. General Information
B. Memory Requirements
C. Loading BASIC
D. Modes of Operation
E. Using the Keyboard
F. Using the Video Display

Chapter 2. BASIC Concepts
A. Programs
B. Statements
C. Data
  1. Data Storage Types
    a. Numeric (Integer, Single and Double Precision)
    b. String
  2. Data Constants
    a. Type Determination
  3. Variables
    a. Names
    b. Types of Variables
      i. Default types
      ii. Tags (!,#,%,$)
    c. Arrays
  4. Data Conversion
D. Operations
  1. Statements
  2. Expressions
  3. Operators
    a. Arithmetic
    b. Logical, Relational and Boolean
    c. String
    d. Evaluation of Expressions
      i. Parentheses
      ii. Order of Operations
      iii. Type Conversions
  4. Functions



For More Information

If you are a newcomer to BASIC, you'll probably need a good programming manual to use along with this book. Here are a few we recommend:

COMPUTER PROGRAMMING IN BASIC FOR EVERYONE, Thomas Dwyer and Michael Kaufman, Radio Shack Catalog Number 62-2015.

BASIC AND THE PERSONAL COMPUTER, Thomas Dwyer and Margot Critchfield; Addison-Wesley Publishing Company, 1978.

BASIC FOR HOME COMPUTERS: A SELF-TEACHING GUIDE, Bob Albrecht, LeRoy Finkel, and Jerald R. Brown; Wiley & Sons, 1978.

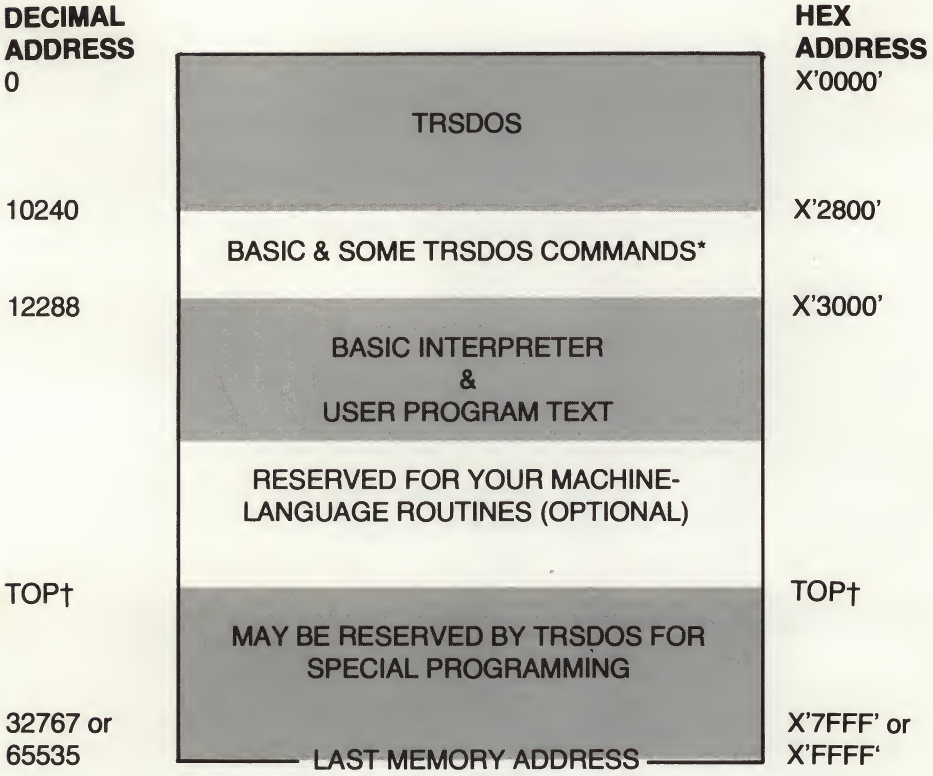
BASIC FROM THE GROUND UP, David E. Simon; Hayden Book Company, 1978.

ILLUSTRATING BASIC, Donald Alcock; Cambridge University Press, 1977.

Memory Requirements

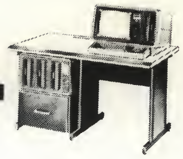
BASIC occupies 14 granules (17920 bytes) on the System diskette. It loads into memory starting at the beginning of user memory, 10240. The amount of memory required by BASIC depends on how many concurrent data files you specify when you load BASIC. During loading, you can also reserve a portion of high memory for storage of machine-language subroutines.

Here's a memory allocation map:


DECIMAL ADDRESS
0
10240
12288
TOP†
32767 or 65535
HEX ADDRESS
X'0000'
X'2800'
X'3000'
TOP†
X'7FFF' or X'FFFF'
TRSDOS
BASIC & SOME TRSDOS COMMANDS*
BASIC INTERPRETER & USER PROGRAM TEXT
RESERVED FOR YOUR MACHINE-LANGUAGE ROUTINES (OPTIONAL)
MAY BE RESERVED BY TRSDOS FOR SPECIAL PROGRAMMING
LAST MEMORY ADDRESS

*Certain TRSDOS commands use memory in the range [X'2800',X'2FFF']. See "Library Commands" in the **TRSDOS Reference Manual** for a list. All TRSDOS commands except for these can be called from BASIC via the BASIC command, SYSTEM.

†TOP is a memory protect address set by TRSDOS. If TRSDOS is not protecting high memory, then TOP is the same as LAST MEMORY ADDRESS.



Loading BASIC

See the **Operation Manual** for instructions on connection, power-up and inserting the System diskette.

Note: Be sure all drives are empty when you turn the Computer on or off. A System diskette must be in Drive 0 (the built-in unit) while the Computer is on. (If you are going to save or change data on a diskette, cover the write-protect notch.)

After the System starts up, it will prompt you to enter the date. Type in the date in MM/DD/YYYY form and press ENTER . For example:

```
07/25/1979  ENTER
```

for July 25, 1979.

Next the System will prompt you to enter the time. **To skip this question,** press ENTER . The time will start at 00:00:00.

To set the time, type in the time in HH.MM.SS 24-hour form. Periods are used instead of colons, since they're easier to type in. The seconds .SS are optional. For example:

```
14.30 ENTER
```

for 2:30 PM.

The System will record the date and time internally and return with the message:

```
TRSDOS READY
```

……

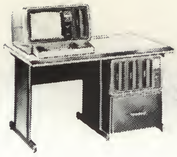
You can now load and execute BASIC. The simplest way to do this is to type:

```
BASIC ENTER
```

BASIC will load (takes several seconds) and display a start-up heading like this:

```
TRS-80 Model II BASIC Vers X.Y
Copyright 1979 by Tandy Corp. Licensed from Microsoft
Created: 16-Aug-79
xxxxx Bytes free, 0 Files
Ready
>
```

XXXXX Bytes free tells you how much memory is available for storage and execution of BASIC programs. 0 files tells you that no data files can be Opened from BASIC. If you want to Open data files, you need to specify how many when you load BASIC (see next paragraphs).



Options for Loading BASIC

There are several other ways to start up BASIC, as summarized in this block:

BASIC *program* −F:*files* −M:*address*

program is a TRSDOS file specification for a BASIC program. After start-up, BASIC will run it. If *program* is omitted, BASIC will start-up in the command mode.

−F:*files* tells BASIC the maximum number of files that may be Open at once. *files* is a number from 0 to 15. If −F:*files* is omitted, maximum is set to 0.

−M:*address* tells BASIC not to use memory above *address*. *address* is a decimal number. If −M:*address* is omitted, BASIC uses all memory up to TOP.

The options allow you to specify any or all of the following:

- A program to run after BASIC is started.
- Maximum number of data files that may be Open at once. The larger the number of files, the less area available for storing and executing your programs. (Each file you specify takes 834 bytes of memory.) So use the smallest value that will suit your needs.
- Highest address to be used by BASIC during program execution. Omit this unless you are going to call machine-language subroutines.

Examples

```
TRSDOS READY
BASIC
```

Tells BASIC not to run a program, but to enter the command mode; to allow for zero concurrent files; and to use all memory available from TRSDOS.

```
TRSDOS READY
BASIC -F:1
```

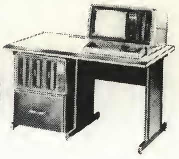
Just like the preceding example, except that only one file can be Open at any given time.

```
TRSDOS READY
BASIC -M:32000
```

BASIC won't allow you to Open any files, and 32000 is the highest address it will use during program execution.

```
TRSDOS READY
BASIC PAYROLL -F:3
```

BASIC will start up, load and run the BASIC program PAYROLL; three data files can be Opened, and BASIC can use all memory available from TRSDOS.



Modes of Operation

BASIC has three modes of operation:

- Command mode – for typing in program lines and immediate lines
- Execute mode – for execution of programs and immediate lines
- Edit mode – for editing program and immediate lines

Command Mode

Whenever you enter the command mode, BASIC displays a header and a special prompt:

```
Ready (header)
> ■     (prompt followed by blinking block "cursor")
```

While you are in the command mode, BASIC will display the prompt at the beginning of the current logical line (the line you are typing in).

A logical line is a string of up to 255 characters and is always terminated with a carriage return (stored when you press **ENTER**). A physical line, on the other hand, is one line on the Display. A physical line contains a maximum of 80 characters.

For example, if you type 100 R's and then press **ENTER** , you will have two physical lines, but only one logical line.

The blinking block is called a cursor. It tells you where the next character you type will be displayed.

In the command mode, BASIC does not take your input until you complete the logical line by pressing **ENTER** . This is called "line input", as opposed to "character input".

Interpretation of an Input Line

BASIC always ignores leading spaces in the line – it jumps ahead to the first non-space character. If this character **is not** a digit, BASIC treats the line as an immediate line. If it *is* a digit, BASIC treats the line as a program line.

For example:

```
Ready
PRINT "THE TIME IS " TIME$ ENTER
```

BASIC takes this as an immediate line.

If you type:

```
Ready
10 PRINT "THE TIME IS " TIME$ ENTER
```

BASIC takes this as a program line.

Immediate Line

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press **ENTER**. For example:

```
Ready
CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is an immediate line. When you press **ENTER**, BASIC executes it.

Program Line

A program line consists of a line number in the range [0,65529], followed by one or more statements separated by colons. When you press **ENTER**, the line is stored in the program text area of memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

```
100 CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is a program line. When you press **ENTER**, BASIC stores it in the program text area. To execute it, type:

```
RUN ENTER
```

Special Keys in the Command Mode

? When used in an immediate line, the question mark can stand for the commonly used keyword PRINT. For example, the immediate line:

```
? "HELLO."
```

is the same as the immediate line:

```
PRINT "HELLO."
```

Note: L? does *not* mean LPRINT.

This abbreviation can be used in a program, too.

. The period can stand for "current program line", i.e., the last program line entered or edited. The period can be used in most places where a line number would normally appear. For example, the immediate line:

```
LIST.
```

tells BASIC to list the current program line.

This abbreviation can be used in a program, too.

(Special Keys in the Command Mode, continued)

' The single-quote tells BASIC to ignore the rest of the logical line. It is an abbreviation for the BASIC keyword REM. When used in a multi-statement line, it does not have to be preceded by a colon. For example, when you type in the line:

```
PRINT 1+1 '2+2
```

BASIC will Print the sum 1+1 but not 2+2.

This abbreviation can be used in a program, too.

Execute Mode

Whenever BASIC is executing statements (immediate lines or programs) it is in the execute mode. In this mode, the contents of the Video Display are under program control.

Special Keys in Execute Mode

HOLD Pauses execution. Press again to continue.

BREAK Terminates execution and returns you to the command mode.

Edit Mode

BASIC includes a line editor for correcting command or program lines. You can also use it to correct keyboard input to an INPUT statement.

To edit an immediate line, press **F1** **before** you have pressed **ENTER** . To edit a program line, type in the command:

EDIT *line number*

where *line number* specifies the desired line.

When the editor is working on a program line, it displays the number of the line being edited. When the editor is working on an immediate line or a line being **input to** an INPUT statement, it displays a ! symbol in the first column on the line.

In the edit mode, Keyboard input is character-oriented, rather than line-oriented. That is, BASIC takes characters as soon as they are typed in—without waiting for you to press **ENTER** .

See **Using the Line Editor** for details.

Using the Keyboard

BASIC has two ways of inputting data from the keyboard:

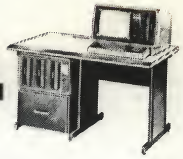
- Line Input—BASIC does not take the input until you press ENTER .
- Character Input: BASIC takes a specified number of characters without waiting for you to press ENTER .

In the Command Mode, BASIC uses line input. In the Edit Mode, it uses character input. Both types of input are available in the Execute Mode. See INPUT, INPUT$, LINE INPUT, INKEY$.

Keyboard Line Input

When you type number, letter, and punctuation keys, BASIC inputs them into the current line. Certain other keys and key combinations have special meanings to BASIC. Control keys not mentioned below are ignored during line input.

BACKSPACE	Backspaces the cursor, erasing the preceding character in the line. Use this to correct typing errors. CTRL H is the same code.
SPACE BAR	Enters a blank space character and advances the cursor.
F1	Puts you in the Edit Mode. The current line will be edited. See **Using the Line Editor.** CTRL A is the same code.
BREAK	Interrupts line entry and starts over with a new line. CTRL C is the same code. BREAK is echoed to the Display as ^C.
TAB	Advances the cursor to the next 8-character boundary. Tab positions are at 0,8,16,24,... Use this for indenting program lines. CTRL I is the same code.
CTRL J	Line feed—starts a new physical line without ending the current logical line.
CTRL O	Toggles (switches the state of) the Display function, i.e., turns it on or off. If the Display is on, CTRL O turns it off. Subsequent characters typed will not be echoed to Display, but will be input into the current line. Any programmed output to the Display will also be ignored. If the Display function is off, CTRL O turns it on. Subsequent characters typed will be echoed to the Display.



(CTRL O , continued)

	Whenever BASIC enters the Command Mode, it turns on the Display function. CTRL O is echoed as ^O.
CTRL R	Retypes the current logical line.
CTRL U	Restarts the current logical line (though the old line remains on the Display). The key is echoed to the display as ^U.
ENTER	Ends the current logical line. BASIC will take the line.
REPEAT	For convenience when you want to repeat a single key, hold down REPEAT while pressing the desired key. For example, to backspace halfway across the Display, hold down REPEAT and BACKSPACE .

Keyboard Character Input

In this mode, key input is not echoed to the display. Any key you press is accepted as input, except for BREAK , which interrupts the input and returns you to the Command Mode, and CTRL O , which toggles the Display function.

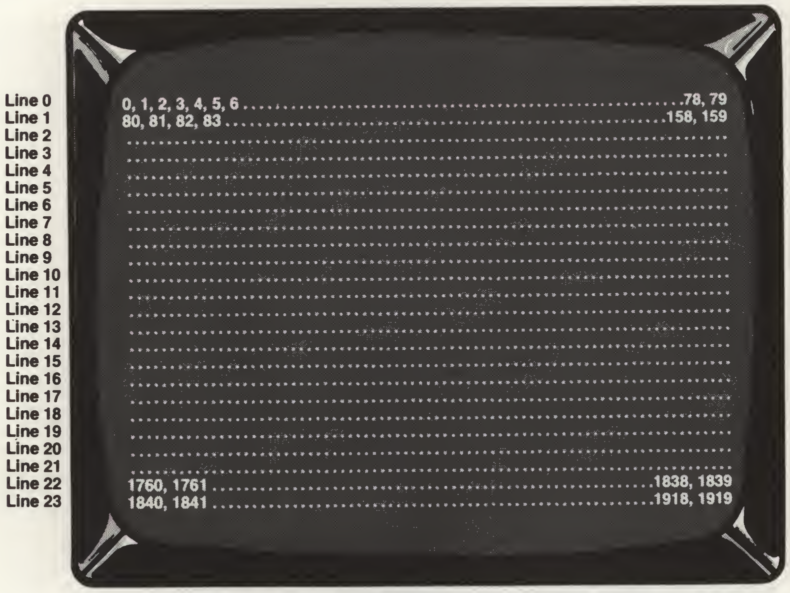
Using the Video Display

Model II BASIC gives you easy access to the Video Display's full character set, including all standard ASCII symbols and 32 special graphics codes. Every character can also be displayed in reverse (black on white).

The Display has two modes of operation–Scroll and Graphics. Cursor motion and position-labeling are different in the two modes. ASCII characters (in the range [32,127]) are always printed in the scroll mode. Graphics characters [128-159] are always printed in the graphics mode.

Scroll Mode

In the Scroll Mode, the Display can be thought of as a sequence of 1920 display positions, as illustrated below:



DISPLAY POSITIONS, SCROLL MODE

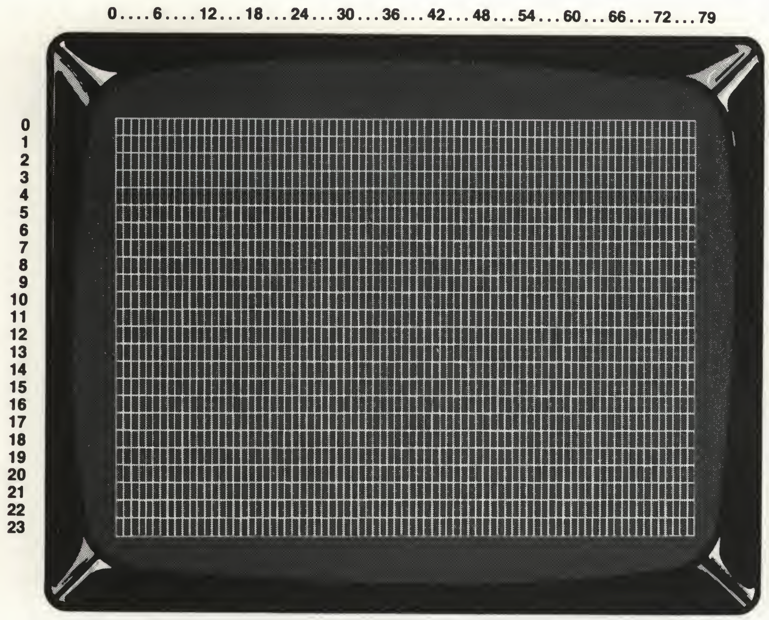
In Scroll Mode output, each time an acceptable display character is received, it is displayed at the current cursor position, and the cursor advances to the next higher numbered position.

When the cursor is on the bottom line and a line-feed or carriage return is received, or when the bottom line is filled, the entire Display is "scrolled":

- Line 0 is deleted
- Lines 1-23 are moved up one line
- Line 23 is blanked
- The cursor is set to the beginning of line 23.

Graphics Mode

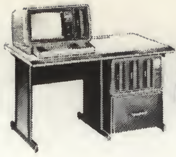
In the Graphics Mode, the Display can be thought of as an 80 by 24 matrix, as illustrated below:



DISPLAY POSITIONS, GRAPHICS MODE

In Graphics Mode output, the cursor "wraps" the display whenever it moves beyond the row or column boundaries. That is:

Current position	Direction	New position
column 79	forward	column 0, same row
column 0	back	column 79, same row
row 23	down	row 0, same column
row 0	up	row 23, same column



Video Display Output

All output to the Display is done via PRINT statements. **To send actual codes to the Display, use the CHR$ function.**

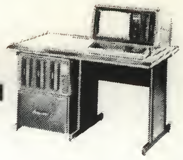
For example:

```
PRINT CHR$(26)
```

Sends code 26 to the Display, which sets the reverse mode.

The table below summarizes the Model II BASIC Display codes.

CODE		DISPLAY FUNCTION
DECIMAL		
1	01	Turns on cursor
2	02	Turns off cursor
4	04	Turns on steady (non-blinking cursor)
8	08	Backspaces cursor and erases
9	09	Tabs cursor to next 8-character boundary
10	0A	Line feed. Moves cursor down one row without changing column position.
13	0D	Moves cursor to start of next line.
23	17	Erases to end of line, cursor doesn't move.
24	18	Erases to end of screen, cursor doesn't move.
25	19	Sets normal (white on black) display mode.
26	1A	Sets reverse (black on white) display mode.
27	1B	Erases screen and homes cursor (position 0).
28	1C	Scroll mode cursor motion: Moves cursor back one position; if old position = 0, cursor doesn't move.
29	1D	Scroll Mode cursor motion: Moves cursor forward one position; if old position = 1919, display is scrolled up one line and new position = 1840.
30	1E	Clears display and sets 80 character-line mode.
31	1F	Clears display and sets 40 character-line mode.
252	FC	Graphics Mode cursor motion: Moves cursor back one column; column = column-1. If column=0, new column=79. Row is unchanged.
253	FD	Graphics Mode cursor motion: Moves cursor forward one column; column = column+1. If column=79, new column=0.
254	FE	Graphics Mode cursor motion: Moves cursor up one row; row=row-1; if row=0, new row=23.
255	FF	Graphics Mode cursor motion: Moves cursor down one row; row=row+1; if row=23, new row=0.



Graphics Characters are codes 128-159. To see them, run the program:

```
10 FOR I=128 TO 159
20       PRINT I; CHR$(I),
30 NEXT
```

Standard ASCII characters (upper and lowercase letters, numbers and punctuation) are codes 32 to 127. To see them, run the program:

```
40 FOR I=32 TO 127
50       PRINT I; CHR$(I),
60 NEXT
```

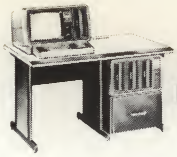
Note: You can print ASCII characters [A-Z, a-o] in the graphics mode by using codes 160-239. Letters p through z cannot be printed in the graphics mode.

For example, the following program turns the Video Display into a simple "page" editor. No scrolling is done, except when a letter from p to z is printed at position (23, 79).

This program also shows how to translate the cursor motion keys ↑ ↓ ← → into codes that will produce the desired result. Run it and try all the cursor control keys. Also try:

CTRL Z	Switch to reverse mode (black on white)
CTRL Y	Switch to normal mode (white on black)
CTRL X	Erase to end of screen
CTRL W	Erase to end of line
ESC	Clear screen
F1	Turn on cursor
F2	Turn off cursor

```
100 A$=INPUT$(1)
110 A=ASC(A$)
120 IF 27<A AND 32 >A THEN A=A+224: GOTO 150    'cursor motion
130 IF A>31 AND A<ASC("p") THEN A=A+128: GOTO 150  'graphics mode
140 IF A=13 THEN PRINT CHR$(255);:PRINT@ (ROW(0),0), "": GOTO 100
150 PRINT CHR$(A);
160 GOTO100
```



Programming the F1 and F2 Keys

The F1 key outputs a code 1; F2 outputs a code 2. If you want these keys to serve a special purpose in a program, you must use character input techniques (INKEY$ and INPUT$) rather than line input (INPUT and LINE INPUT). Once your program has received a character, it can check to see whether the character is a 1 or a 2. If it *is* one of these codes, the program can perform whatever function you choose to associate with the F1 or F2 key.

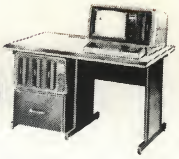
The following program shows a typical use of F1 and F2 keys. Lines 330, 340 and 350 input a keyboard character and take appropriate action if the character was a 1 F1 or a 2 F2.

```
100 ON ERROR GOTO 400
110 CLS
120 PRINT "PRESS <S> FOR SQUARE ROOTS"
130 PRINT "PRESS <C> FOR CUBE ROOTS"
140 PRINT "PRESS <L> FOR LOGARITHMS"
150 PRINT "PRESS <A> FOR ANTI-LOGARITHMS"
160 PRINT "PRESS <Q> TO QUIT"
170 A$ = INPUT$(1)
180 IF A$ = "S" THEN 240
190 IF A$ = "C" THEN 250
200 IF A$ = "L" THEN 260
210 IF A$ = "A" THEN 270
220 IF A$ ="Q" THEN END
230 GOTO 170
240 DEFFN M(N) = SQR(N): H$ = "SQUARE ROOT": GOTO 280
250 DEFFN M(N) = N^(1/3): H$ = "CUBE ROOT": GOTO 280
260 DEFFN M(N) = LOG(N): H$ = "NATURAL LOG": GOTO 280
270 DEFFN M(N) = EXP(N): H$ = "NATURAL ANTI-LOG"
280 K=0: CLS: PRINT @ (2,10),H$
290 PRINT @ (4,0), CHR$(24)
300 K = K+1: PRINT TAB(10); K; TAB(20); FNM(K)
310 IF ROW(0) < 20 THEN 300
320 PRINT @ (22,0), "PRESS <F1> FOR MORE, <F2> FOR MENU"
330 A$=INPUT$(1)
340 IF A$=CHR$(1) THEN 290
350 IF A$=CHR$(2) THEN 110
360 GOTO 330
400 IF ERR = 6 THEN LINE INPUT "OVERFLOW--PRESS <ENTER>";X$:RESUME 100
410 ON ERROR GOTO 0
```

Chapter 2

BASIC Concepts

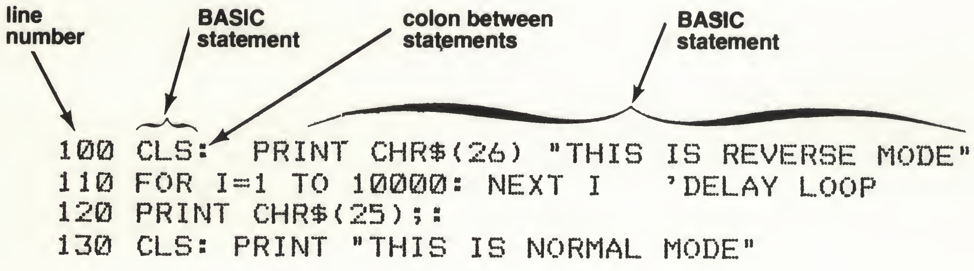
This chapter contains the background information you'll need to write programs in Model II BASIC. It describes the types of data (information) BASIC can handle, and the operations BASIC can perform on the data.



Programs

A program consists of one or more numbered logical lines, each line consisting of one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. The program lines can include up to 255 total characters not including the line number, and may be broken into two or more physical lines.

For example, here is a program:



```
100 CLS:  PRINT CHR$(26) "THIS IS REVERSE MODE"
110 FOR I=1 TO 10000: NEXT I   'DELAY LOOP
120 PRINT CHR$(25);:
130 CLS: PRINT "THIS IS NORMAL MODE"
```

When BASIC executes a program, it handles the statements one at a time, starting at the first and proceeding to the last. Some statements allow you to change this sequence. (See "Program Sequence Statements".)

Statements

A statement is a complete instruction to BASIC, telling the Computer to perform some operations. If the operations involve data, the statement may include that, too. For example,

```
PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is a complete statement. The number 2 is the data, and the operations are:

- Displaying the message in quotes
- Computing the square root of 2
- Displaying the resultant value

Data

BASIC can handle two kinds of data:

- Numbers, representing quantities and subject to standard mathematical operations
- Strings, representing sequences of characters and subject to special non-mathematical string operations

Each kind of data has its own memory storage requirement and its own range of values.

Numeric Data

BASIC allows three types of numbers: integer, single-precision and double-precision. You can declare the type of a number, or let BASIC assign a type. Each type serves a specific purpose in terms of precision, speed and arithmetic operations, and range of possible values.

Integer Type (Speed and Efficiency, Limited Range)

To be stored as an integer type, a number must be whole and in the range [-32768,32767]. An integer value requires two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:
1   32000   -2   500   -12345
can all be stored as integers.

Single-Precision Type (General Purpose, Full Numeric Range)

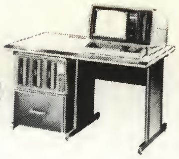
Single-precision numbers can include up to 7 significant digits, and can represent normalized values* with exponents up to ± 38, i.e., numbers in the range:

$[-1 \times 10^{38}, -1 \times 10^{-38}]\ [1 \times 10^{-38}, 1 \times 10^{38}]$

A single-precision value requires 4 bytes of memory for storage. BASIC assumes a number is single-precision if you do not specify the level of precision.

---

*In this reference manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.23 × 10.



For example:
10.001 -200034 1.774E6 6.024E-23 123.4567
can all be stored as single-precision values.

Note: When used in a decimal number, the symbol E stands for "single-precision times 10 to the power of..." Therefore 6.024E-23 represents the single-precision value:

6.024×10^{-23}

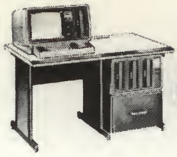
Double-Precision Type (Maximum Precision, Slowest in Computations)

Double-precision numbers can include up to 17 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires 8 bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single-precision or integer.

For example:
1010234578 -8.7777651010 3.1415926535897932 8.00100708D12
can all be stored as double-precision values.

Note: When used in a decimal number, the symbol D stands for "double-precision times 10 to the power of..." Therefore 8.00100708 D12 represents the value

$8.00100708 \times 10^{12}$



Summary of Numeric Data Types

Data Type	Range	Storage Requirement* (Bytes)	Typical Values
Integer	[-32768,+32767]	2	-1, 0, 1, 3000, -125 225, -1001
Single-Precision	[$-1*10^{+38}$, $-1*10^{-38}$] [$+1*10^{-38}$, $+1*10^{+38}$] Up to seven significant digits (Prints only six).	4	1,2345, 22.50, -100.001, 2.1415 3.14159, 1.545E5
Double-Precision	[$-1*10^{+38}$, $-1*10^{-38}$] [$+1*10^{-38}$, $+1*10^{+38}$] Up to 17 significant digits (Prints only 16).	8	123000.00 3.1415926535897932 45.500101D-18 1.66666667D-5

***For each variable you define, an additional three bytes are used as overhead.**

String Data

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. Any ASCII character can be stored in a string. For example, the data:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage. The above string would be stored internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

Data Constants

All data is input to a program in the form of constants—values which are not subject to change. For example, the statement:

```
PRINT "1 PLUS 1 EQUALS" 2
```

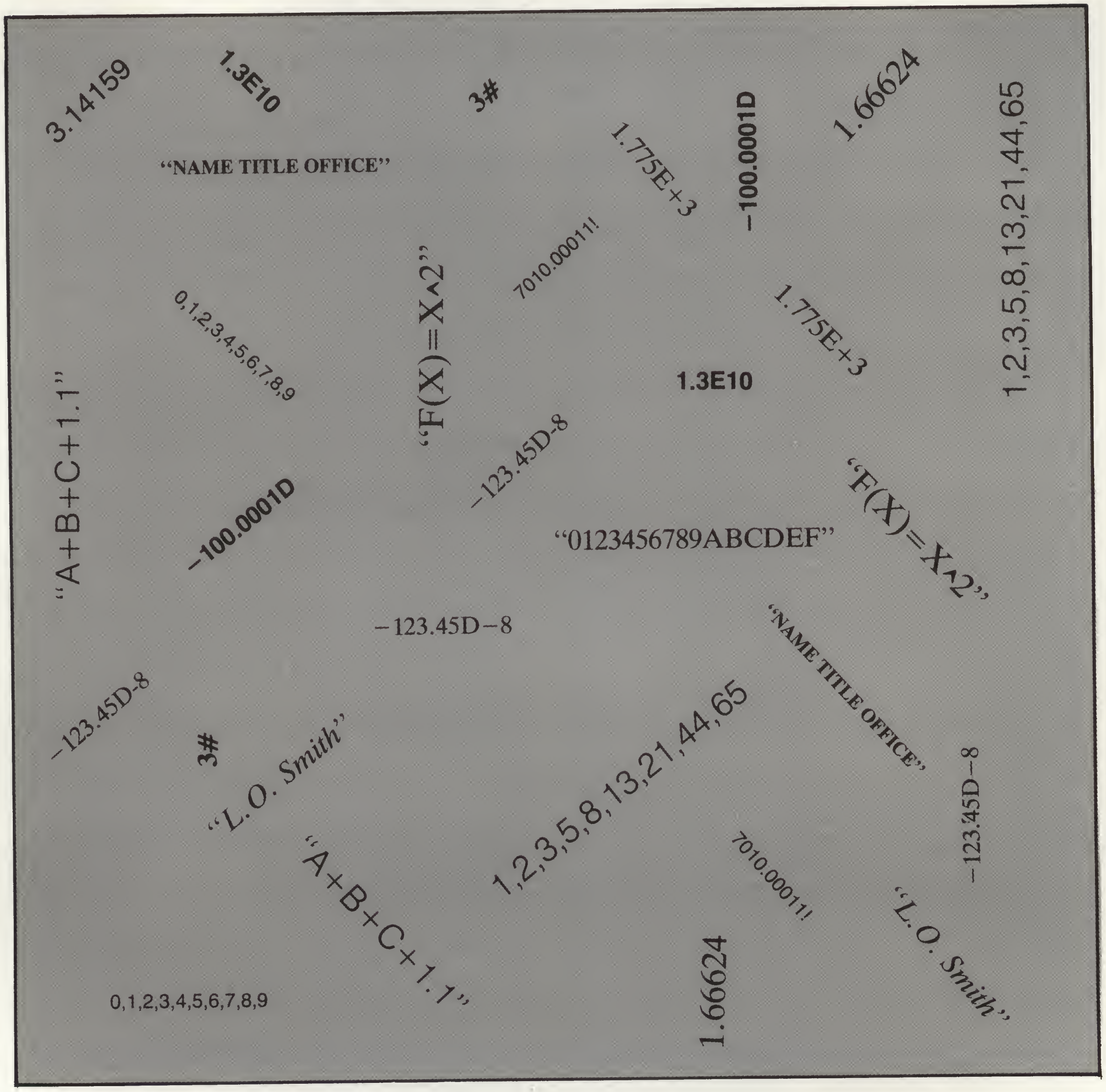
contains one string constant,

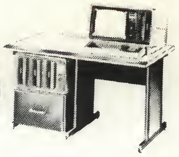
```
1 PLUS 1 EQUALS
```

and one numeric constant,

```
2
```

In this example, the constants serve as "input" to the PRINT statement—telling it what values to print on the Display.





Type Determination

When BASIC encounters a data constant in a statement, it must determine the type of the constant (string, integer, single-precision or double-precision). Here are the rules it uses:

Rule I. If the value is enclosed in double-quotes, it is a string. For example, in the statements:

```
A$="YES"
B$="3331 Waverly Way"
PRINT "1234567890"
```

the values in quotes are automatically categorized as strings. (A$ and B$ are variables, as explained later in this section.)

Rule II. If the value is not in quotes, it is a number. There are exceptions to this rule. See DATA, INPUT, LINE INPUT, INKEY$, and INPUT$. For example, in the statements:

```
A=123001
B=1
PRINT 12345, -7.32145E6
```

all the data is numeric.

Rule III. Whole numbers in the range [-32768, 32767] are integers. For example, the statements:

```
A=12350
B=-12
PRINT 10012, -21000
```

contain integer constants only.

Rule IV. Numbers which are not integer type and which contain seven or fewer digits are single-precision. For example, in the statements:

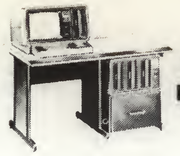
```
A=1234567
B=-1.23
PRINT 11000.25, 1.3321
```

all the numbers are single-precision.

Rule V. If the number contains more than seven digits, it is double-precision. For example, in the statements:

```
A=12345678901234567
B=-1000000000000.1
PRINT 2.777000321
```

all the numbers are double-precison. (The *variables* A and B may be double-precision, single-precision, or integer, as explained later on. Only the *constants* are described here.)



Type Declaration Tags

You can override BASIC's normal typing criteria by adding the following "tags" to the end of the numeric constant:

! Makes the number single-precision. For example, in the statement:

```
A=12.345678901234!
```

the constant is classified as single-precision, and shortened to seven digits: 12.34567

E Single-precision exponential format. The E indicates the constant is to be multipled by a specified power of 10. For example:

```
A=1.2E5
```

stores the single-precision number 120000 in A.

Makes the number double-precision. For example, in statement:

```
PRINT 3#/7
```

the first constant is classified as double-precision before the division takes place.

D Double-precision exponential format. The D indicates the constant is to be multipled by a specified power of 10. For example:

```
A=1.23456789D-1
```

The double-precision constant has the value 0.123456789.

Hexadecimal and Octal Constants

Model II BASIC allows two additional types of constants: hexadecimal and octal numbers.

Hexadecimal numbers are quantities represented in base 16 notation, composed of the numerals 0-9 and the letters A-F. Hexadecimal constants must be in the range [0,FFFF]. They are stored as two-byte integers, corresponding to decimal integers as follows:

Hexadecimal Range	Equivalent Decimal Range
[0,7FFF]	[0,32767]
[8000,FFFF]	[-32768,-1]

Any number preceded by the symbol &H is interpreted as a hexadecimal constant. For example:

```
     &HA010  &HFE    &HD1    &HC     &H0D    &H4000
```

are all hexadecimal constants.

Octal numbers are quantities represented in base 8 notation, composed of the numerals 0-7. Octal constants must be in the range [0,177777]. They are stored as two-byte integers, corresponding to decimal integers as follows:

Octal Range	Equivalent Decimal Range
[0,77777]	[0,32767]
[100000,177777]	[-32768,-1]

Any number preceded by the symbol &O or & is interpreted as an octal constant. For example:

```
    &O70      &O44      &1777     &7170     &17        &O1234
```

are all octal constants.

Variables

A variable is a place in memory—a sort of box or pigeonhole—where data can be stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities.

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by a digit, 0 through 9, or another letter.

For example:

```
     A          AA          A2          B7          MJ
```

are all valid and distinct variable names.

Variable names may be longer than two characters. However, only the first two characters are significant in BASIC.

For example:

```
     SU          SUM          SUPERNUMERARY
```

are all treated as the **same** variable by BASIC.

Reserved Words

Certain combinations of letters are reserved as BASIC keywords, and cannot be used in variable names. For example:

```
     OR          LAND          NAME          LENGTH     MIFFED
```

cannot be used as variable names, because they contain the reserved words OR, AND, NAME, LEN, and IF, respectively.

See the **Appendix** for a list of reserved words.

Types of Variables

As with constants, there are four types of variables. The first three are numeric: integer, single-precision and double-precision; the fourth is string.

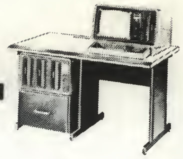
Depending on its type, one variable can contain values from only one of these groups.

The first letter of the variable name determines what the type is. Initially, all letters A through Z have the single-precision attribute. This means that all variables are single-precision (that is, they can only hold single-precision values).

For example:

```
     A          B          X1          CY          TRS          H4
```

are all single-precision variables initially.



However, you can assign different attributes to any of the letters, by means of DEFINT (define-integer), DEFDBL (define double-precision), and DEFSTR (define-string) statements. DEFSNG A-Z (define single-precision) is assumed unless other DEF statements are used.

For example:

```
DEFSTR L
```

makes all variables which start with L into string variables. After the above statement, the variables:

```
L    LP    LAST
```

can all hold **string values only.**

Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags for variables:

%	Integer
!	Single-precision
#	Double-precision
$	String

For example:

```
I%        FT%       NUM%      COUNTER%
```

are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N and C.

```
T!        RY!       QUAN!     PERCENT!
```

are all single-precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q and P.

```
X#        RR#       PREV#     LASTNUM#
```

are all double-precision variables, **regardless** of what attributes have been assigned to the letters X, R, P and L.

```
Q$        CA$       WRD$      ENTRY$
```

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W and E.

Note that any given variable name can represent four different variables. For example:

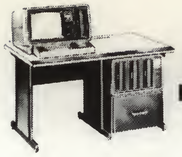
```
A5#       A5!       A5%       A5$
```

are all valid and **distinct** variable names.

One further implication of type declaration: Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR  C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.



Array Variables

BASIC allows subscripted variables or arrays. An array name references a list of values, or elements, instead of a single element.

The array can have one or more dimensions. Each dimension is specified by a subscript. Array subscripts **always** start with zero. Therefore the statement:

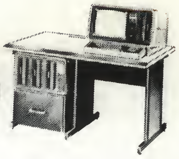
```
    DIM A(12,10)
```

creates an array A with 13 rows (numbered 0-12) of 11 columns (0-10), for a total of (13*11=143) elements.

```
    A(5,7)
```

refers to the element at row 5, column 7 in array A.

See the DIM statement description for more information.



Data Conversion

Often it is necessary to convert a value from one type to another type. BASIC will perform many conversions automatically; other conversions require that you use special functions.

For example, suppose you want to add two numbers:

```
1 + 1.2345678901234567
```

The first number is an integer constant; the second, a double-precision constant. Because of different storage formats for the two types, the operation is physically impossible until one of the numbers is converted to match the other's type.

According to rules described later, BASIC converts the 1 to double precision. Then the two double-precision numbers can be added to produce a double-precision result.

What concerns us here is not the addition, or the rule for deciding which number is converted. Here we are only interested in *the conversion itself*.

Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$=1234
A#="1234"
```

are illegal. (Use STR$ and VAL to accomplish such conversions.)

Legal Conversions

BASIC can convert any numeric type into any other numeric type. For example:

```
A#=A%              'integer to double-precision
A!=A#              'double-precision to single-precision
A!=A%              'integer to single-precision
```

Rules for Conversion

Single or double-precision to integer type

BASIC returns the largest integer that is not greater than the original value.

Note: The original value must be greater than or equal to -32768, and less than 32768.

Examples

```
A%=-10.5
```

Assigns A% the value -11.

```
A%=32767.9
```

Assigns A% the value 32767.

```
A%=2.5D3
```

Assigns A% the value 2500.

```
A%=-123.45678901234578
```

Assigns A% the value -124.

```
A%=-32768.1
```

Produces an Overflow Error (out of integer range).

Integer to single- or double-precision

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal place.

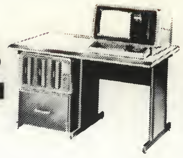
Examples

```
A#=32767
```

Stores 32767.000000000000 in A#.

```
A!=-1234
```

Stores -1234.000 in A!.



Double- to single-precision

This involves converting a number with up to 17 significant digits into a number with no more than seven. BASIC chops off (truncates) the least significant digits to produce a seven-digit number. Before Printing such a number, BASIC rounds it off (4/5 rounding) to six digits.

Examples

```
A!=1.234567890124567
```

Stores 1.234567 in A! However, the statement:

```
PRINT A!
```

will display the value 1.23457, because only six digits are displayed. The full seven digits are stored in memory.

```
A!=1.3333333333333333
```

Stores 1.333333 in A!.

Single- to double-precision

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error will be introduced. For example:

```
A#=1.5
```

Stores 1.5000000000000 in A#, since 1.5 *does* have an exact binary representation.

However, for numbers which have no exact binary representation, an error is introduced when zeros are added. For example:

```
A#=1.3
```

Stores 1.299999952316284 in A#.

Because most fractional numbers do not have an exact binary representation, you should keep such conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double-precision:

```
A#=1.3#        A#=1.3D
```

Both store 1.3 in A#.

Here is a special technique for converting single-precision to double-precision, without introducing an error into the double-precision value. It is useful when the single-precision value is stored in a variable.

Take the single-precision variable, convert it to a string with STR$, then convert the resultant string back into a number with VAL. That is, use:

VAL (STR$ (*single-precision variable*))

For example, the following program:

```
10 A!=1.3
20 A#=A!
30 PRINT A#
```

prints a value of:

```
1.299999952316284
```

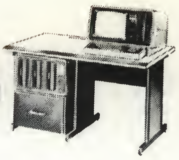
Compare with this program:

```
10 A!=1.3                'single-precision
20 A#=VAL(STR$(A!))      'special conversion technique
30 PRINT A#
```

which prints a value of:

```
1.3
```

The conversion in line 20 causes the value in A! to be stored accurately in double-precision variable A#.



Operations

An operation instructs the Computer to do something.

There are four levels of operations:
- Statements, which are complete instructions
- Expressions, which serve as parameters and data for statements
- Operators, which act on one or two data elements ("operands") and are used in expressions
- Functions, which act on one or more data elements ("arguments") and are also used in expressions

Statements

Statements tell the Computer to perform some action. Statements are complete in themselves. Once the statement has been written, no other information needs to be added to the statement for it to be executed.

For example, the statement:

```
DEFINT N-R
```

is complete as it stands.

A statement is made up of a keyword* followed by whatever parameters or data are needed. The data is usually represented by an expression (defined below).

For example:

```
PRINT "MODEL II"
```

Tells BASIC to display the message inside quotes. PRINT is the keyword; "MODEL II" the data.

```
LIST 100-130
```

Tells BASIC to list the resident program lines in the range 100-130. LIST is the keyword; 100-130, the parameter.

```
A1 = 5 * A / 3
```

Tells BASIC to give A1 the value of the expression on the right of the equals sign.

---

*A keyword is any sequence of characters which has a predefined meaning for BASIC. PRINT , INPUT , and SQR are all examples of keywords.

Expressions

The concept of an expression is important in this manual, since it is used in most of the syntax descriptions. Throughout these descriptions, you will encounter the terms *numeric expression*, *string expression*, *logical expression*, etc. Understanding the concept will allow you to grasp the full potential of BASIC's operations.

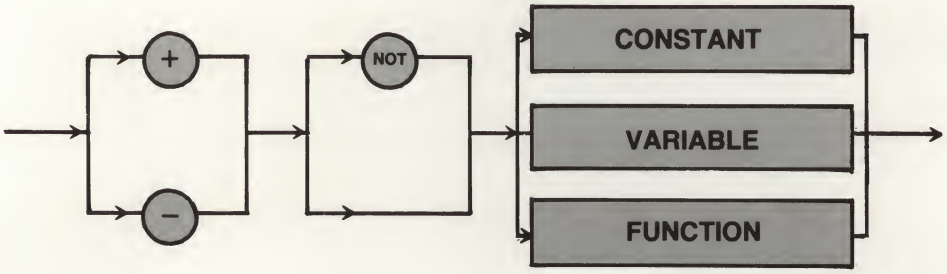
Expressions are composed of:
- Constants
- Variables
- Operators
- Functions

A **simple expression** consists of a single term: a constant, variable, or function preceded by an optional + or − sign or the logical operator NOT.

For example:

```
+ A               3.3               -5               NOT A
```

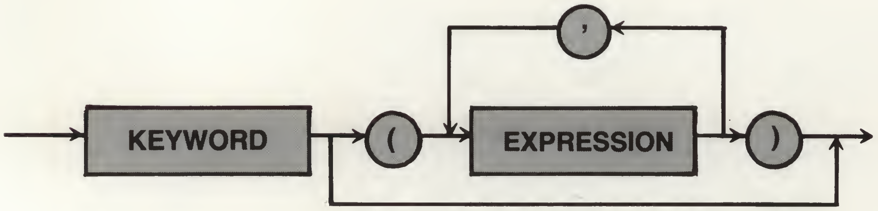
Here's how a **term** is formed (items in square boxes are defined elsewhere):



A **function** consists of a keyword usually followed by an argument list in parentheses. Each of the arguments can be an expression. For example:

```
SIN(33)          SQR(A)          CHR$(34)          ABS(A-B)
```

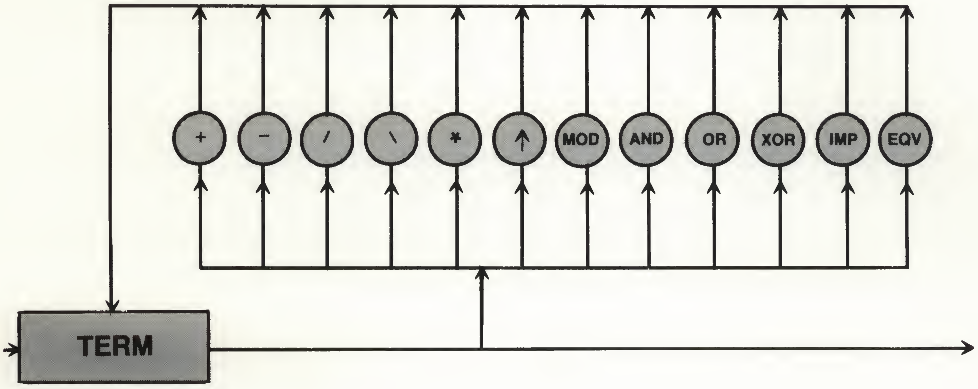
Here's how a **function** is formed:

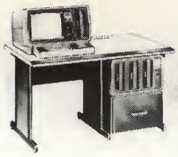


In general, an expression consists of one term or two or more terms combined by operators (defined below). For example:

```
A-1       X+3.2-Y     A/3 * (LOG(Y))   NOT(A OR B)
```

Here's how a complex expression is formed:





Operators

An operator is a single symbol or word which signifies some action to be taken on one or two specified values referred to as operands.

In general, an operator is used like this:

operand-1 operator operand-2

> *operand-1* and -2 can be expressions. A few operators take only one operand, and are used like this:

operator operand

> This is the form for a unary operation

Examples:

```
6 + 2
```

The addition operator + connects or relates its two operands 6 and 2 to produce the result 8.

```
-5
```

The negation operator – acts on a single operand 5 to produce the result negative 5.

Neither 6+2 nor –5 can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
A = 6+2
PRINT -5
```

Operators fall into three categories:
- Numeric
- Logical
- String

based on the kinds of operands they require and the results they produce.

Numeric Operators

In the descriptions below, we use the terms **integer** operation, **single-precision** operation and **double-precision** operation. Integer operations involve two-byte operands; single-precision, four-byte operands; and double-precision, eight-byte operands. It's very important to be aware of what precision will be used in a given operation, since **the more bytes involved, the slower the operation**.

There are nine different numeric operators. Two of them, sign + and sign −, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT   -77; +77
```

the sign operators − and + produce the values negative 77 and positive 77, respectively.

Note: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands. These operators are

+	Addition
−	Subtraction
*	Multiplication
/	Division
\	Integer division (keyboard character CTRL 9)
^	Exponentiation (keyboard character SHIFT 6)
MOD	Modulus arithmetic

Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single precision, the integer is converted to single-precision and four-byte addition is done. When one operand is single-precision and the other is double-precision, the single-precision number is converted to double-precision and eight-byte addition is done.

Examples:

```
PRINT 2+3
```

Integer addition.

```
PRINT 3.1 + 3
```

Single-precision addition.

```
PRINT 1.2345678901234567 + 1
```

Double-precision addition.

Subtraction

The – operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 - 11
```

Integer subtraction.

```
PRINT 33 -11.1
```

Single-precision subtraction.

```
PRINT 12.345678901234567 - 11
```

Double-precision subtraction.

Multiplication

The * operator is the symbol for multiplication. Once again, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 * 11
```

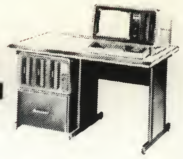
Integer multiplication.

```
PRINT 33 * 11.1
```

Single-precision multiplication.

```
PRINT 12.345678901234567 * 11
```

Double-precision multiplication.



Division

The / symbol is used to indicate ordinary division. Both operands are converted to single or double-precision, depending on their original precision:

- If either operand is double-precision, then both are converted to double-precision and eight-byte division is performed.
- If neither operand is double-precision, then both are converted to single-precision and four-byte division is performed.

Examples:

```
    PRINT 3/4
```

Single-precision division.

```
    PRINT 3.8 / 4
```

Single-Precision division.

```
    PRINT 3 / 1.2345678901234567
```

Double-precision division.

Integer Division

The integer division operator \ ("backslash") converts its operands into integer type, then performs integer division, in which the remainder after division is ignored, leaving an integer result. (If either operand is outside the range [-32768,32767], an error will occur.)

Note: To enter the \ operator, press **CTRL 9**

For example:

```
  PRINT 7 \ 3
```

prints the value 2, since 7 divided by 3 equals 2 remainder 1.

Exponentiation

The symbol ^ ("circumflex") denotes exponentiation. It converts both its operands to single-precision, and returns a single-precision result.

Note: To enter the ^ operator, press **SHIFT 6** .

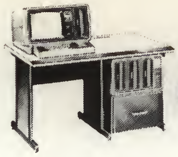
For example:

```
  PRINT 6 ^.3
```

prints 6 to the .3 power.

Modulus Arithmetic

The MOD ("modulo") operator allows you to do modulus arithmetic, i.e., arithmetic in which every number is converted to its equivalent in a cyclical counting scheme. For example, a 24-hour clock indicates the hour in modulo 24; although the hour keeps incrementing, it is always expressed as a number from 0 to 23.



MOD requires two operands, for example:

```
A MOD B
```

B is the modulus (the counting base) and A is the number to be converted.

(Expressed in mathematical terms, A MOD B returns the **remainder** after whole-number division of A by B. In this sense, it is the converse of \, which returns the **whole number quotient** and ignores the remainder.)

MOD converts both operands to integer type before performing the operation. If either operand is outside the range [-32768,32767] an error will occur.

Examples:

```
PRINT 155 MOD 15
```

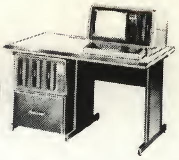
Prints 5, since 155/15 gives a whole number quotient of 10 with remainder 5.

```
PRINT 79 MOD 12
```

Prints 7, since 79/12 equals 6 with remainder 7.

```
10 INPUT "TYPE IN AN ANGLE IN DEGREES"; A%
20 PRINT A% "=" A% \ 90 "* 90 +" A% MOD 90
```

Input a positive angle greater than 90. Line 20 expresses the angle as a multiple of 90 degrees plus a remainder.



The table below summarizes the precision of operations for all numeric operators.

(I=integer, S=single-precision, D=double-precision.)

Operator(s)	Precision of Operand(s)	Precision of Value Returned
+, -, *	I *op* I I *op* S S *op* S I *op* D S *op* D D *op* D	I S D
/	I *op* I I *op* S S *op* S I *op* D S *op* D D *op* D	S D
\	All possible combinations	I
^	All possible combinations	S
MOD	All possible combinations	I
+ (sign) – (sign)	*op* I *op* S *op* D	I S D

Important: For effects of conversions on accuracy, see "Data Conversion".

Logical Operators

Logical operators deal with true/false conditions, comparisons, and tests. They allow you to build elaborate decision-making structures into programs, to perform bit manipulations, to sort data, etc. An expression involving a logical operator is called a **logical expression**.

All logical operators convert their operands to two-byte integers. If an operand is outside of the range [-32768, 32767] an error will occur.

The logical operators include the three relational operators: <, >, =; and six Boolean word-operators: AND, OR, XOR, NOT, IMP, EQV

Relational operators compare two operands for numerical precedence. Here is a table of the relational operators and their various combinations:

Relational Operators

<	Less than
>	Greater than
=	Equal to
>< or <>	Not equal to
=<or<=	Less than or equal to
=>or>=	Greater than or equal to

Relational operators can return only two possible values: true or false. Actually, BASIC returns the number -1 to indicate true, and 0 to indicate false. But the quantity (-1 or 0) is rarely used as a number. More often, it is used as a decision-making operator, as in the line:

```
IF A=B THEN 1000 ELSE END
```

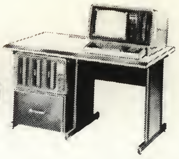
The logical expression A = B returns negative one (-1) when A equals B, and zero when A does not equal B. But you don't care about the numbers -1 and 0. What matters to you is that if the expression is true, control branches to line 1000; otherwise BASIC ends the program.

Here's an example where the result of a logical expression *is* used as a quantity:

```
MAX = -(A<B)*B - (B<=A)*A
```

For any two integer-type values A and B, MAX contains the larger of the two.

Note: All relational operators can also be used to compare strings for precedence. The result of such a comparison is still either a true (logical -1) or false (logical 0). See "String Operators".



Boolean Operators

In this section, we will explain how Boolean operators are implemented in Model II BASIC. However, we will not try to explain Boolean algebra, decimal-to-binary conversions or binary arithmetic. If you need to learn something about these topics, Radio Shack's **Understanding Digital Electronics** (Catalog Number 62-2010) and **TRS-80 Assembly-Language Programming** (62-2006) are the books to start with.

Model II BASIC includes six Boolean operators:

AND OR XOR EQV IMP NOT

All the Boolean operators relate two operands, except for NOT, which acts on a single operand.

These operators can be used to set up decision structures. For this application, both operands are usually relational expressions, like (A < B), and the operator is one of the following: AND, OR, XOR, NOT.

AND

If both expressions are true, then AND returns a logical true. Otherwise it returns a logical false. For example:

```
IF A=B AND B<0 THEN 100
```

OR

If either of the expressions is true, or both are true, this operand returns a logical true. Otherwise it returns a logical false. For example:

```
IF GAME=OVER OR TIME>=LATE THEN END
```

XOR ("Exclusive–OR")

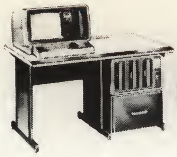
Only when **one** of the expressions is true (but not both) does XOR return a logical true. Otherwise it returns a logical false. For example:

```
IF JOHN=OUT XOR JOE=OUT THEN PRINT "ONLY ONE IS HERE"
```

NOT

NOT is a unary operator (acts on one operand). When the expression is true, NOT returns a logical false. When it is false, NOT returns a logical true. For example:

```
IF NOT (A>B) THEN PRINT "A IS NOT GREATER THAN B"
```



Bit Manipulation

For this application, both operands are usually numeric expressions. BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

Note: The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

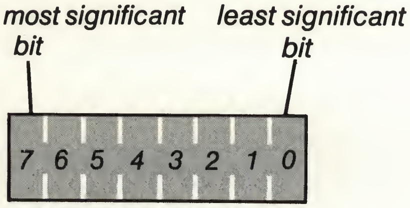
The following table summarizes the action of Boolean operators in bit manipulation.

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits are 1, the result will be 1. Otherwise, the result will be 0.	1 1 0 0	1 0 1 0	1 0 0 0
OR	Result will be 1 unless both bits are 0.	1 1 0 0	1 0 1 0	1 1 1 0
XOR	Result will be 1 unless both bits are the same.	1 1 0 0	1 0 1 0	0 1 1 0
EQV	Result will be 1 unless both bits are different.	1 1 0 0	1 0 1 0	1 0 0 1
IMP	Result will be 1 unless first bit is 1 and second bit is 0.	1 1 0 0	1 0 1 0	1 0 1 1
NOT	Result is opposite of bit.	1 0		0 1

As an example of bit manipulation, suppose you want to change lowercase characters to uppercase and vice-versa. You could do this by checking the ASCII code of each character (See table in the appendix) and adding or subtracting decimal 32 (hexadecimal 20) depending on whether the character was uppercase or lower. But this routine could be done more simply, using only the operator XOR.

The ASCII codes for uppercase characters are decimal 65-90 (hexadecimal 41-5A); for lowercase, decimal 97-122 (hexadecimal 61-7A). Looking at these ranges in binary, you can see that all capital letters have a 0 in bit position 5, while all lowercase letters have a 1 in bit position 5.

Note: Position 7 is the most significant bit; position 0 is least significant, as illustrated below:



One byte

So, to convert from lower to uppercase and vice versa, you just toggle (reverse the state of) bit 5. Decimal 32 has the following binary representation:

```
00100000
```

Notice that bit 5 is a 1; all others are zeroes. When you XOR decimal 32 with any number, you will effectively toggle bit 5. For letters, this will switch cases, upper to lower and vice versa.

For instance, since 72 is the ASCII code for "H":

```
PRINT CHR$(72 XOR 32)
```

prints a lowercase "h".

You can check this by consulting XOR in the table above and XOR-ing the two numbers by hand.

Sample Program

```
10 PRINT "This program converts uppercase to lowercase and v.v."
20 A$=INPUT$(1)
30 IF "A" <= A$ AND A$ <= "Z" OR "a" <= A$ AND A$ <= "z"
   THEN A$=CHR$(ASC(A$) XOR 32)
40 PRINT A$;
50 GOTO 20
```

String Operators

There are seven string operators in Model II BASIC. These operators allow you to **compare** strings and to **concatenate** them (i.e., string them together).

Comparison
The comparison operators for strings are the same as those for numbers, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare sorting precedence (i.e., alphabetical sequence).

<	Precedes
>	Follows
=	Has the same precedence
<>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

Comparison is made character by character on the basis of ASCII codes. When a non-matching character is found, the string containing the character with a lower ASCII code is taken as the smaller ("precedent") of the two strings. See the Appendix for an ASCII code table.

Examples:

```
"A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
"CODE" < "COOL"
```

ASCII for O is 79; for D it's 68.

If, while comparison is proceeding, the end of one string is reached before any non-matching characters are found, the **shorter** string is considered to be precedent. For example:

```
"TRAIL" < "TRAILER"
```

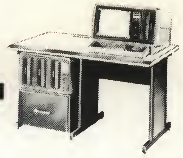
Leading and trailing blanks are significant. For example:

```
" A" < "A "
```

ASCII for "ƀ" (space) is 32; for A it's 65.

```
"Z-80" < "Z-80a"
```

The string on the left is four characters long; the string on the right is five.



Here are some examples of how you might use the string comparison operators in a program:

```
  IF A$<>B$ THEN END
```

If string A$ is not the same as B$, the program ends.

```
  IF A$<B$ THEN PRINT A$
```

If A$ alphabetically precedes B$, A$ is printed.

```
  IF NME$="CARRUTHERS" OR CITY$="BUFFALO" THEN PRINT NME$,CITY$
```

If the value of NME$ is CARRUTHERS, then CARRUTHERS plus the current value of CITY$ will be printed, **or** if the value of CITY$ is BUFFALO, then BUFFALO will be printed plus the current value of NME$.

Concatenation

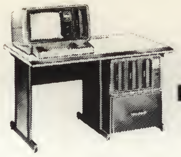
Concatenation ("linking") is represented by the symbol +. This operator takes two strings as its operands and returns a single string as its result by adding the string on the right of the + sign to the string on the left. If the new string is greater than 255 characters, a String Too Long error wll occur.

For example:

```
   PRINT "CATS " + "LOVE " + "MICE"
```

prints:

```
   CATS LOVE MICE
```



Evaluation of Expressions

When an expression involves multiple operations, BASIC performs the operations according to a well-defined hierarchy, so that results are always predictable.

Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expression inside the parentheses before evaluating the rest of the expression. For example, the expression:

```
    8-(3-2)
```

is evaluated like this:

```
    3-2=1
      8-1=7
```

With nested parentheses, BASIC starts evaluation at the innermost level and works outward. For example:

```
    4 * (2-(3-4) )
```

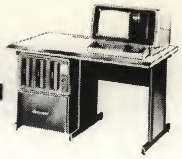
is evaluated like this:

```
    3-4=-1
      2- -1=3
        4 * 3=12
```

Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses the following hierarchy to determine what operation to do first. Operators are shown below in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed as encountered **from left to right**.

^ (Exponentiation)
+, − (Unary sign operands [**not** addition and subtraction])
*, /
\ (Integer division)
MOD
+, − (Addition and subtraction)
<, >, =, <=, >=, <>
NOT
AND
OR
XOR
EQV
IMP



For example, in the line

```
X * X + 5↑2.8
```

BASIC will find the value of 5 to the 2.8 power. Next it will multiply X * X, and finally add this value to the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses, e.g.:

```
X * (X+5↑2.8)
```

or

```
X * (X+5)↑2.8
```

Here's another example:

```
IF X=0 OR Y>0 AND Z=1 THEN 255
```

The relational operators=and >have the highest precedence, so they will be performed first, one after another, left to right. Then the Boolean operations will be performed. AND has a higher precedence than OR, so the AND operation will be performed before the OR. Therefore, the line above means that if X = 0, or if Y > 0 and Z = 1, control branches to line 255.

If the line above looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X=0 OR ((Y>0) AND (Z=1)) THEN 255
```

Type Conversions

During evaluation of an expression, BASIC often has to perform type conversions. Unless you're careful in forming expressions, these conversions can produce invalid results. For example, in the expression:

```
A# * C!
```

C! must be converted to double-precision before the multiplication can take place. This will usually introduce an error into the result.

Before evaluating the expression:

```
A + B↑1.2345678
```

BASIC must convert 1.2345678 to single-precision. **You cannot expect double-precision from a single-precision operator or function.**

See "Data Conversion" for details on the effects of type conversion on accuracy, and for special conversion techniques.

Functions

A function is a built-in subroutine. The functions supplied in Model II BASIC save you from having to write equivalent BASIC routines, and they operate faster than a BASIC routine would.

A function consists of a keyword followed by required input values, referred to as **arguments** or **parameters**. The arguments are always enclosed in parentheses and separated by commas. Some functions have no arguments; others require up to three. The quantity output or returned by a function is called the **value** of the function.

Examples:

```
SQR(A)
```

Tells BASIC to compute the square root of the quantity A. SQR is the keyword, and A is the argument.

```
MID$(A$,3,2)
```

Tells BASIC to return a substring of the string A$, starting with the third character, with length 2. MID$ is the keyword; A$, 3 and 2 are its arguments or parameters.

Since functions are syntactically equivalent to expressions, they cannot stand alone in a BASIC program. They must be used in statements.

For example:

```
A=SQR(B)
```

Assigns A the value of square root of B.

```
PRINT MID$(A$, 3, 2)
```

Prints the substring of A$ starting at the third character and two characters long.

```
PRINT LOG(SQR(2))
```

Prints the natural logarithm of the square root of 2.

In this manual, functions are classified as **numeric** when they return a number, and **string** when they return a string. Wherever the syntax calls for a numeric expression, you can use a numeric function; for a string expression, you can use a string function.

There is another **special class** of functions which return information about the allocation of memory and the location of various quantities in memory. For example:

```
MEM
```

Returns the number of bytes of memory available for storing program text, numeric and array variables.

Chapter 3

BASIC Keywords

To help you grasp the purpose of each of Model II BASIC's many keywords, we have divided them up into several groups, and covered each group separately. Within each group, the keywords appear in alphabetical order. The groups are:

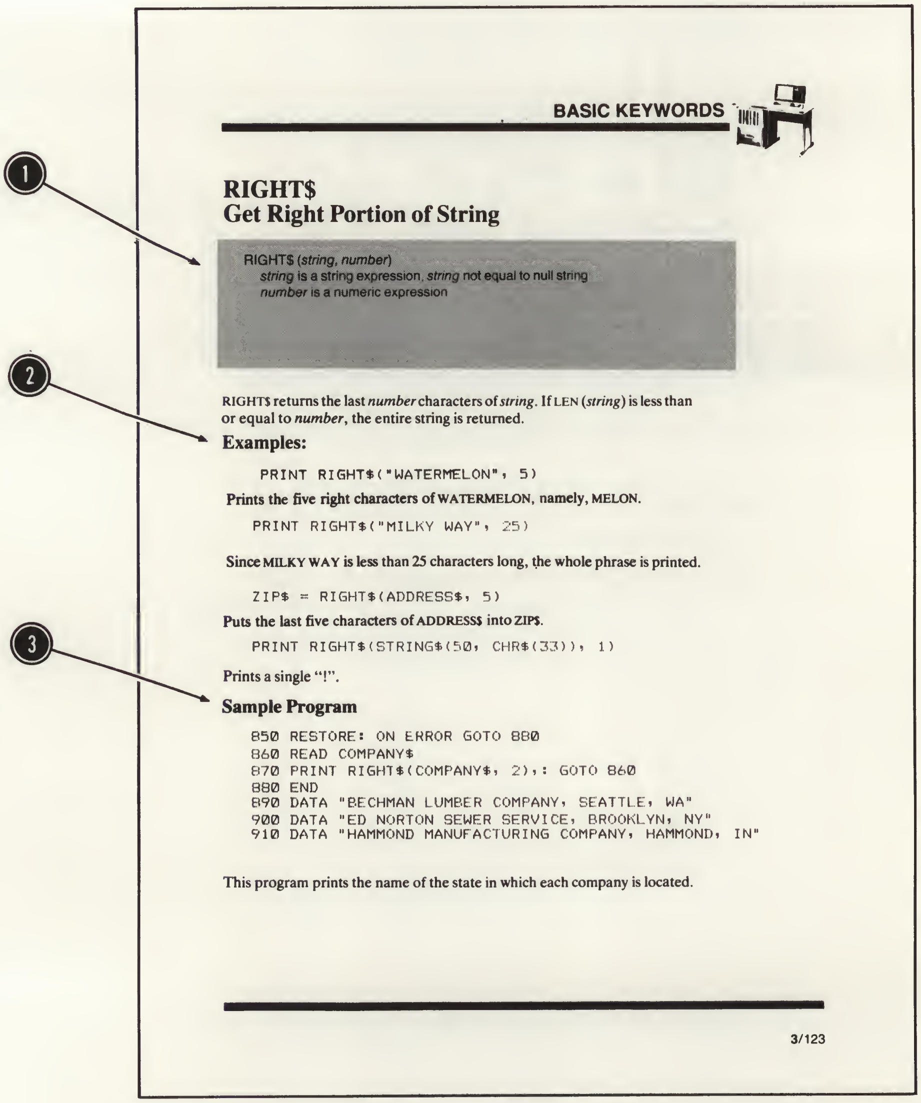
- *Command Statements*
- *Program Statements*
- *Error-Handling Statements*
- *Numeric Functions*
- *String Functions*
- *Input/Output Functions*
- *Special Functions*

Note: For most keyword descriptions, we've provided three different information blocks.

① **Syntax.** This area is highlighted in gray and gives the most general description of how the keyword is used.

② **Examples.** These show specific occurrences of the keyword in a complete BASIC statement. These one-line examples are **not** intended to by typed in and run.

③ **Sample Programs.** Run these for a demonstration of how the keyword works.



BASIC KEYWORDS

RIGHT$
Get Right Portion of String

RIGHT$ (*string, number*)
string is a string expression, *string* not equal to null string
number is a numeric expression

RIGHT$ returns the last *number* characters of *string*. If LEN (*string*) is less than or equal to *number*, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

Prints the five right characters of WATERMELON, namely, MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

Since MILKY WAY is less than 25 characters long, the whole phrase is printed.

```
ZIP$ = RIGHT$(ADDRESS$, 5)
```

Puts the last five characters of ADDRESS$ into ZIP$.

```
PRINT RIGHT$(STRING$(50, CHR$(33)), 1)
```

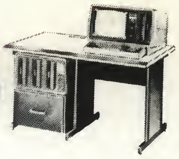
Prints a single "!".

Sample Program

```
850 RESTORE: ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2),: GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMOND MANUFACTURING COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

3/123



Command Statements

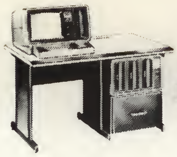
Command statements tell BASIC to enter another operation mode or to perform various System functions (like loading a program from disk). Although they can be included inside a program, **their primary use is outside of a program**.

For example, the command statement

```
NEW
```

Erases the entire program currently in memory and zeroes all variables.

Keyword	Purpose
AUTO	Number lines automatically
DELETE	Erase program lines from memory
EDIT	Edit program line
KILL	Delete a disk file
LIST	List program to display
LLIST	List program to line printer
LOAD	Load program from disk
MERGE	Merge disk program with resident program
NEW	Erase program from RAM
RENUM	Renumber program
RUN	Execute program
SAVE	Save program on disk
SYSTEM	Return to TRSDOS



AUTO
Number Lines Automatically

AUTO *startline, increment*
startline is a line number specifying the first line number to be used. If *startline* is omitted, 10 is used. A period (".") can be substituted for *startline.* In this case, the current line number is used. If *startline* is omitted and a comma is used, *startline* is set to 0.
increment is a number specifying the increment to be used between lines. If *increment* is omitted, 10 is used.

AUTO turns on an automatic line numbering function. After you enter this command, BASIC will supply the *line number*. All you have to do is type in the text of the line and press **ENTER** . Then AUTO will display the next line number, using *increment* or a default increment of 10.

To turn off the AUTO function, press **BREAK** at any time. The current line will be cancelled.

Whenever AUTO provides a line number that is already in use, it will display an asterisk immediately after the line number. Press **BREAK** if you do not want to change that line.

Examples

```
AUTO
```

starts automatic numbering with line 10, using increments of 10 between line numbers.

```
AUTO 100
```

starts numbering with 100, using increments of 10 between line numbers.

```
AUTO 1000, 100
```

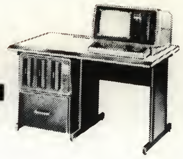
starts numbering with 1000, using increments of 100 between line numbers.

```
AUTO ,5
```

starts numbering with 0, using increments of 5 between line numbers.

```
AUTO.
```

starts numbering with the current line number, using increments of 10 between line numbers.



DELETE
Erase Program Lines from Memory

DELETE *startline-endline*

startline is a line number specifying the lower limit for the deletion. If *startline* is omitted, then the first line in the program is used as *startline*.

endline is a line number specifying the last line in your program that you want to delete. *endline* must reference an existing program line.

A period (".") can be substituted for either *startline* or *endline*. The period signifies the current line number.

DELETE removes from memory the specified range of program lines.

Examples

```
DELETE 70
```

Erases line 70 from memory. If there is no line 70, an error will occur.

```
DELETE 50-110
```

Erases lines 50 through 110 inclusive.

```
DELETE -40
```

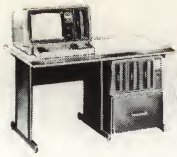
Erases all program lines up to and including line 40.

```
DELETE -.
```

Erases all program lines up to and including the line that has just been entered or edited.

```
DELETE.
```

Erases the program line that has just been entered or edited.



EDIT
Edit Program Line

EDIT *line number*

EDIT allows the specified line to be revised without affecting any other lines. The EDIT command has a powerful set of subcommands which are discussed in detail in Chapter 5.

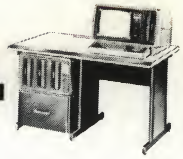
Examples

```
EDIT 100
```

Edits line 100

```
EDIT.
```

Edits the current line.



KILL
Delete File from Disk

KILL *file*

file is a string expression defining a TRSDOS file specification.
If *file* is a constant, it must be enclosed in quotes.

KILL deletes the specified file from the diskette directory.

If no drive specification is included in the file specification BASIC will search for the first drive that contains the file, and attempt to delete it.

Do not Kill an open file. Close it first.

Example

```
KILL "FILE/BAS"
```

deletes this file from the first drive which contains it.

```
KILL "DATA:2"
```

deletes this file from drive 2 only.

LIST
Display Program Lines

LIST *startline-endline*

startline is a line number specifying the lower limit for the listing. If *startline* is omitted, then the first line in the program is used.

endline is a line number specifying the upper limit for the listing. If *endline* is omitted, the last line in the program is used.

A period (".") can be substituted for either *startline* or *endline*. The period signifies the current line number.

LIST instructs the Computer to display the specified range of program lines currently in memory. The arguments are optional.

Examples

```
LIST
```

Displays the entire program. To stop the automatic scrolling, press HOLD. This will freeze the display. Press any key to continue the listing.

```
LIST 50
```

Displays line 50.

```
LIST 50-85
```

Displays lines in the range 50-85.

```
LIST 227-
```

Displays line 227 and all higher-number lines.

```
LIST.-
```

Displays the program line that has just been entered or edited, and all higher-numbered lines.

```
LIST-227
```

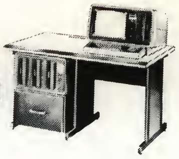
Displays all lines up to and including 227.

```
LIST-.
```

Displays all lines up to and including the line that has just been entered or edited.

```
LIST.
```

Displays the line that has just been entered or edited.



LLIST
Print Program Lines

LLIST *startline-endline*

startline is a line number specifying the lower limit for the listing. If *startline* is omitted, then the first line in the program is used as *startline*.

endline is a line number specifying the upper limit for the listing. If *endline* is omitted, the last line in the program is used as *endline*. A period (".") can be substituted for either *startline* or *endline*. The period signifies the current line number.

LLIST works like LIST, but its output is to the Printer rather than the Display. LLIST instructs the Computer to print the specified range of program lines currently in memory. The arguments are optional.

Examples

```
LLIST
```

Lists the entire program to the printer. To stop this process, press **HOLD**. This will cause a temporary halt in the Computer's output to the Printer. Press any key to continue printing.

```
LLIST 780
```

Prints line 780.

```
LLIST 68-90
```

Prints lines in the range 68-90.

```
LLIST 50-
```

Prints lines 50 and all higher-numbered lines.

```
LLIST.-
```

Prints the program line that has just been entered or edited plus all higher-numbered lines.

```
LLIST-50
```

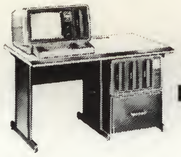
Prints all lines up to and including 50.

```
LLIST-.
```

Prints all lines up to and including the line that has just been entered or edited.

```
LLIST.
```

Prints the line that has just been entered or edited.



LOAD
Load Basic Program File

LOAD *file*, R
file is a string expression defining a TRSDOS file specification.
If *file* is a constant, it must be enclosed in quotes.
R (optional) tells BASIC to RUN the program after it is loaded.

This command loads a BASIC program file into RAM. If the R option is used, BASIC will proceed to RUN the program automatically. Otherwise, BASIC will return to the command mode.

LOAD wipes out any resident BASIC program, clears all variables, and Closes all Open files unless the R option is used, in which case open files will not be closed.

LOAD with the R option is equivalent to the command RUN *file*, R. Either of these commands can be used inside programs to allow program chaining (one program calling another). Files currently Open are **not** Closed.

If you attempt to LOAD a non-BASIC file, a Direct Statement in File error will occur.

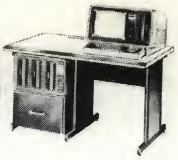
Example

```
LOAD "PROG1/BAS:2"
```

This loads PROG1/BAS from drive 2. BASIC then returns to the command mode.

```
LOAD "PROG1/BAS"
```

Since no drive specification is included in this command, BASIC will begin searching for this program file, starting with drive zero.



MERGE
Merge Disk Program with Resident Program

MERGE *file*

file is a string expression specifying a BASIC file in ASCII format, e.g., a program saved with the A option.

If *file* is a constant, it must be enclosed in quotes.

The MERGE statement takes a BASIC program from disk and merges it with the resident BASIC program in RAM.

Program lines in the disk program are inserted into the resident program in sequential order. For example, if three of the lines from the disk program are numbered 75, 85, and 90, and three of the lines from the resident program are numbered 70, 80, and 100, when MERGE is used on the two programs, this portion of the new program will be numbered 70, 75, 80, 85, 90, 100.

If line numbers in the disk program coincide with line numbers in the resident program, the resident lines will be replaced by those from the disk program. For example, if three of the lines from the disk program are numbered 5, 10, and 20, and three of the lines from the resident program are numbered 10, 20, and 30, when MERGE is used on the two programs, this portion of the new program will be numbered 5, 10, 20, 30. Lines 10 and 20 of the new program will be identical to lines 10 and 20 on the disk program.

MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

Example

Let's say we have a BASIC program on disk, PROG2/TXT, which we want to merge with the program we've been working on in RAM. Then we use:

```
MERGE "PROG2/TXT"
```

Sample Uses

MERGE provides a convenient means of putting program modules together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in RAM:

```
80 REM      MAIN PROGRAM
90 GOSUB 1000
100 REM      PROGRAM LINE
110 REM      PROGRAM LINE
120 REM      PROGRAM LINE
130 END
```

And suppose the following subroutine, SUB/TXT, is stored on disk in ASCII format:

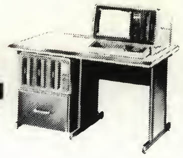
```
1000 REM      BEGINNING OF SUBROUTINE
1010 REM      SUBROUTINE LINE
1020 REM      SUBROUTINE LINE
1030 REM      SUBROUTINE LINE
1040 RETURN
```

We can MERGE the subroutine with the main program using the statement

```
     MERGE "SUB/TXT"
```

and the new program in RAM would be:

```
80 REM      MAIN PROGRAM
90 GOSUB 1000
100 REM      PROGRAM LINE
110 REM      PROGRAM LINE
120 REM      PROGRAM LINE
130 END
1000 REM      BEGINNING OF SUBROUTINE
1010 REM      SUBROUTINE LINE
1020 REM      SUBROUTINE LINE
1030 REM      SUBROUTINE LINE
1040 RETURN
```



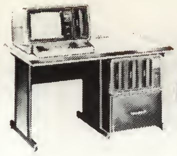
NEW
Erase Program from Memory

```
NEW
```

NEW erases all program lines, sets numeric variables to zero and string variables to null.

Example

```
NEW
```



RENUM
Renumber Program

RENUM *newline, startline, increment*

newline specifies the new line number of the first line to be renumbered. If *newline* is omitted, the line number 10 is used.

startline specifies the line number in the original program where you want to start renumbering. If *startline* is omitted, the entire program will be renumbered.

increment specifies the increment to be used between each successive renumbered line. If *increment* is omitted, 10 is used.

RENUM changes all line numbers in the specified range, as well as all line number references appearing after GOTO, GOSUB, THEN, ON . . . GOTO, ON . . . GOSUB, ON ERROR GOTO, and ERL [relational operator] – throughout the program.

All the RENUM arguments are optional.

Examples

```
RENUM
```

Renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

```
RENUM 6000, 5000, 100
```

Renumbers all lines numbered from 5000 up. The first renumbered line will become 6000, and an increment of 100 will be used between subsequent lines.

```
RENUM 10000, 1000
```

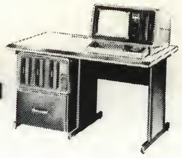
Renumbers line 1000 and all higher-numbered lines. The first renumbered line will become line 10000. An increment of 10 will be used between subsequent line numbers.

```
RENUM 100,,100
```

Renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that the commas must be retained even though the middle argument is gone.

```
RENUM,,5
```

Renumbers the entire program, starting with a new line number of 10, and incrementing by 5's.



Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10, 20 and 30, then the command:

```
RENUM 15,30
```

   is illegal, since the result would be to move the third line of the program ahead of the second. In this case, an FC (illegal function call) error will result, and the original program will be left unchanged.

2. RENUM will not create new line numbers greater than 65529. Instead, an FC error will result, and the original program will be left unchanged.

3. If an undefined line number is used inside your original program, RENUM will print a warning message, UNDEFINED LINE XXXX in YYYY, where XXXX is the original line number reference and YYYY is the original number of the line containing XXXX.

   Note that RENUM will renumber the program in spite of this warning message. It will replace the number XXXX with 5 blanks, and will renumber YYYY, according to the parameters in your RENUM command.

   For example, if your original program includes the line:

```
110 GOTO 1000
```

   but does **not** include a line 1000, then RENUM will print a warning,

```
UNDEFINED 1000 IN 110
```

   and renumber the program. The text of original line 110 will be changed to

```
GOTO
```

RUN
Execute Program

RUN *startline*
startline is a line number specifying where you want program execution to start. If *startline* is omitted, the first line in the program is used.

RUN *file, R*
file is a string expression specifying a BASIC program stored on disk.
If *file* is a constant, it must be enclosed in quotes.
R is an optional parameter. If used, BASIC leaves all previously Open files Open. If omitted, BASIC Closes all Open files.

RUN followed by a line-number or nothing at all simply executes the program in memory, starting at the specified line or at the beginning of the program.

RUN followed by a file specification loads a program from disk and then runs it. Any resident BASIC program will be replaced by the new program.

RUN automatically CLEARS all variables.

Examples

```
RUN
```

Execution starts at lowest line number.

```
RUN 100
```

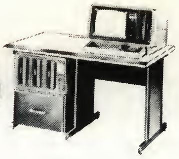
Execution starts at line 100.

```
RUN "PROGRAM/A"
```

When you type the above line and press **ENTER** , the specified BASIC program will be loaded and executed.

```
RUN "EDITDATA",R
```

Loads and executes EDITDATA, leaving Open files Open.



Sample Uses

Suppose you have two programs in memory. One of them begins at line 100 and ends at line 180; the other begins at 200 and ends at 350. Furthermore, the first program has been appropriately terminated (i.e., 180 END). You want to run the second program, stop, observe its output, and then run the first. Type:

```
RUN 200
```

and the second program will execute. When you want to begin execution of the first program, simply type:

```
RUN
```

Sample Program

Suppose you save the following program on disk with the name "PROG1/BAS":

```
200 PRINT "PROG1 EXECUTING..."
210 RUN "PROG2/BAS"
```

And save this program on disk with the name "PROG2/BAS":

```
220 PRINT "PROG2 EXECUTING..."
230 RUN "PROG1/BAS"
```

Now type:

```
RUN "PROG1/BAS"
```

and you'll see a simple example of program chaining. Hold down the **BREAK** key to interrupt the program chain.

SAVE
Save Program in a Disk File

SAVE *file*, A
  file is a string expression for a TRSDOS file specification.
  If *file* is a constant, it must be enclosed in quotes.
  ,A causes the file to be stored in ASCII format. If omitted, a compressed format is used.

The SAVE command lets you save your BASIC programs on disk. If the file you use as the argument of SAVE already exists, its contents will be lost as the file is re-created.

You can save a program in compressed or ASCII format. Using compressed format takes up less disk space and is faster during SAVEs and LOADs. BASIC programs are stored in RAM using compressed format.

Using the ASCII option makes it possible to do certain things that can't be done with compressed-format BASIC files. For example:

- A disk file must be in ASCII form before the MERGE command can be used.
- Programs which read in other programs as data typically require that the data programs be stored in ASCII.

For compressed-format programs, a useful convention is to use the extension /BAS. For ASCII-format programs, use /TXT.

Examples

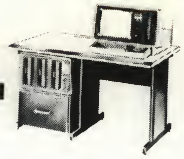
```
SAVE "FILE1/BAS.JOHNQDOE:3"
```

saves the resident BASIC program in compressed format. The file name is FILE1; the extension is /BAS; the password is JOHNQDOE. The file is placed on drive 3.

```
SAVE "MATHPAK/TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non-write-protected drive.

Note: BASIC compressed-format files have a record length of 256. This record length allows fastest saving, loading and copying (TRSDOS COPY command). BASIC ASCII-format files (SAVE with A option) have a record length of 1).



SYSTEM
Return to TRSDOS

SYSTEM *command*

command is a string expression specifying a TRSDOS command.

If *command* is a constant, it must be enclosed in quotes.

command **must not** specify any of the TRSDOS "high memory commands" listed in the **TRSDOS Reference Manual,** Library Commands section. Furthermore, to call DEBUG from BASIC, you **must** turn DEBUG on before starting BASIC.

SYSTEM is used to return to TRSDOS, the disk operating system. The argument *command* causes the System to execute the specified TRSDOS command and immediately return back to BASIC. Your program and variables will be unaffected.

If *command* is omitted, SYSTEM returns you to the TRSDOS READY mode.

Examples

```
SYSTEM
```

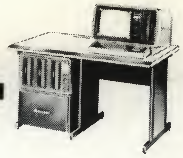
Returns you to TRSDOS. Your resident BASIC program will be lost.

```
SYSTEM "DIR"
```

Causes the TRSDOS command, DIR (print directory) to be run, and then returns to BASIC. Your resident BASIC program will remain intact.

Sample Program

```
350 PRINT "THIS IS A PROGRAM FILE."
360 PRINT "BEFORE SAVING IT, I WANT TO SEE"
365 PRINT "WHAT FILENAMES HAVE BEEN USED."
370 FOR N = 1 TO 1000: NEXT
380 SYSTEM "DIR"
390 PRINT "NOW I CAN CHOOSE A FILENAME"
395 PRINT "WHICH HASN'T BEEN USED."
400 END
```



Program Statements

These are generally used inside programs, rather than in the command mode. They are divided into four categories:

- Definition and initialization
- Assignment
- Program sequence
- Input/output

Definition and Initialization

These statements perform several functions, including changing the default values set initially by BASIC, and reserving or allocating memory space for your program to use. Such statements generally are used at the beginning of a program (Exceptions: DATA, ERASE, REM, RESTORE).

Keyword	Purpose
CLEAR	Clear variables and allocate string space
DATA	Store program data
DEFDBL	Define variables as double-precision
DEFFN	Define function
DEFINT	Define variables as integers
DEFSNG	Define variables as single-precision
DEFSTR	Define variables as strings
DEFUSR	Define entry point for USR routine
DIM	Dimension an array
ERASE	Erase an array
RANDOM	Reseed random number generator
REM	Comment line (remarks)
RESTORE	Reset DATA pointer

CLEAR
Clear Variables and Allocate String Space

CLEAR *string-space*, *memory-size*
string-space is a numeric expression; if *string-space* is omitted, string allocation is unchanged.
memory-size is a numeric expression. If *,memory-size* is omitted, memory-protection is left unchanged.

When used without an argument, CLEAR sets all numeric variables to zero, and all string variables to null. When used with a single argument, this command performs a second function in addition to the one just described: it causes the Computer to set aside for string storage the specified number of bytes. When BASIC is initialized 100 bytes are automatically set aside for strings.

The amount of string storage CLEARed must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur. By setting string storage to the exact amount needed, your program can make more efficient use of memory. A program which uses no string variables could include a CLEAR 0 statement, for example.

You can also change the memory-protect address, i.e. the highest memory address BASIC will use. This will be useful when you want to load a machine-language routine from BASIC, and have BASIC not use that memory area.

Since CLEAR initializes all variables, you must use it near the beginning of your program, before any variables have been defined and before any DEF statements.

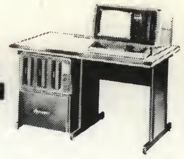
Examples

```
CLEAR
```

All variables are cleared but string space is unchanged.

```
CLEAR 75
```

All variables are cleared and 75 bytes of memory are reserved for string storage.



Sample Program

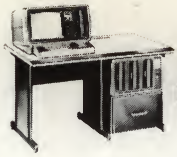
```
60 CLEAR 100
70 PRINT FRE(A$)
80 CLEAR 0
90 PRINT FRE(A$)
100 CLEAR 100
```

```
CLEAR 200, 61000
```

Clears variables, sets string space to 200 bytes, and makes 61000 the highest address BASIC can use to run your programs. Memory contents above this address will not be changed by BASIC. If you loaded BASIC in a 64K RAM Computer but did not reserve any memory, you might like to reserve memory while you are in BASIC. For example, suppose you have a machine-language subroutine stored in a file named USRO/SUB. If the subroutine loads at 61001, then you might put the following lines at the beginning of your program:

```
10 CLEAR 100,61000
20 SYSTEM"USR0/SUB"
30 DEFUSR0=61001
```

After BASIC executes lines 10 and 20, the subroutine will be in memory and will be protected from over-writing by BASIC. Line 30 sets up a USR0 call to the subroutine.



DATA
Store Program-Data

DATA *item-list*
item list is a list of string and/or numeric constants, separated by commas.

The DATA statement lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Expressions are not allowed in a DATA list. If your string values include leading blanks, colons, or commas, you must enclose these values in quotes.

DATA statements may appear anywhere it is convenient in a program. Generally, they are placed consecutively, but this is not required. It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement.

Examples

```
1340 DATA NEW YORK, CHICAGO, LOS ANGELES, PHILADELPHIA, DETROIT
```

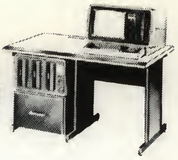
This line contains five string data items. Note that quote marks aren't needed, since the strings contain no delimiters and the leading blanks are not significant.

```
1350 DATA 2.72, 3.14159, 0.0174533, 57.29578
```

This line contains four numeric data items.

```
1360 DATA "SMITH, T.H.", 38, "THORN, J.R.", 41
```

The quote marks are required around the first and third items.



Sample Program

```
170 CLS: PRINT: READ HEADING$: PRINT HEADING$: PRINT STRING
$(42, "-")
180 ON ERROR GOTO 500
190 READ C$: READ DOB: READ N$
200 PRINT C$, DOB, N$: GOTO 190
210 DATA COMPOSER     DATE OF BIRTH     NATIONALITY
220 DATA BOCCHERINI,  1743,             ITALIAN
230 DATA GLUCK,       1714,             GERMAN
240 DATA HAYDN,       1732,             AUSTRIAN
250 DATA MOZART,      1756,             AUSTRIAN
500 IF ERR = 4 THEN END
510 ON ERROR GOTO 0
```

This program prints a list of some major composers of the late 18th Century. Notice we use an ON ERROR GOTO statement to allow the inclusion of data lists of unknown length. For a different means of achieving the same end, see the sample program for READ.

DEFDBL
Define Variables as Double-Precision

DEFDBL *letter list*

letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas. A letter-range is of the form:

letter1 – letter2

DEFDBL causes variables beginning with any letter specified in *letter list* to be classified as double-precision, unless a type declaration character is added to the variable name. Double-precision values include 17 digits of precision, though only 16 are printed out.

DEFDBL is ordinarily used at the beginning of a program. Otherwise, it might suddenly change the meaning of a variable that lacks a type declaration character.

Examples

```
DEFDBL K
```

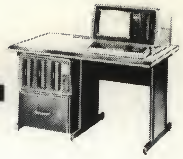
causes any variable beginning with the letter K to be double-precision.

```
DEFDBL Q, S-Z, A-E
```

causes any variable beginning with the letters Q, S through Z, or A through E to be double-precision.

Sample Program

```
570 DEFDBL X
580 A = 3.1415926535897932
590 X = 3.1415926535897932
600 PRINT "PI IN SINGLE PRECISION IS" A
610 PRINT "PI IN DOUBLE PRECISION IS" X
```



DEF FN
Define Function

DEF FN *function name (argument-1, . . .) = formula*
function name is any valid variable name.
argument-1 and subsequent arguments are used in defining what the function does.
formula is an expression usually involving the agrument(s) passed on the left side of the equals sign.

The DEF FN statement lets you create your own function. That is, you only have to call the new function by name, and the associated operations will automatically be performed. Once a function has been defined with the DEF FN statement, you can call it simply by inserting FN in front of *function name*. You can use it exactly as you might use one of the built-in functions, like SIN, ABS and STRING$.

The type of variable used for *function name* determines the type of value the function will return. For example, if *function name* is single precision, then that function will return a single-precision value, regardless of the precision of the arguments.

The particular variables you use as arguments in the DEF FN statement (*argument-1, . . .*) are not assigned to the function. When you call the function later, any variable name of the same type can be used.

Furthermore, using a variable as an argument in a DEF FN statement has no effect on the value of that variable. So you can use that particular variable in another part of your program without worrying about interference from DEF FN.

The function must be defined with at least one argument. In other words, there must be at least one variable in the position of *argument-1* above, even if this variable is not actually used to pass a value to the function.

Examples

```
DEF FNR(A,B) = A + INT((B - (A - 1)) * RND(0))
```

This statement defines function FNR which returns a random number between integers A and B. The values for A and B are passed when the function is "called", i.e., used in a statement like:

```
Y = FNR(R1, R2)
```

If R1 and R2 have been assigned the values 2 and 8, this line would assign a random number between 2 and 8 to Y.

```
DEF FNL$(X) = STRING$(X, "-")
```

Defines function FNL$ which returns a string of hyphens, X characters long. The value for X is passed when the function is called:

```
PRINT FNL$(30)
```

This line prints a string of 30 hyphens.

Here's an example showing DEF FN used for a complex computation – in double precision.

```
DEF FNX#(A#, B#) = (A# - B#) * (A# - B#)
```

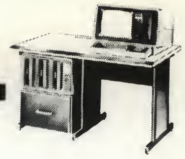
Defines function FNX# which returns the double-precision value of the square of the difference between A# and B#. The values for A# and B# are passed when the function is called:

```
S# = FNX#(A#, B#)
```

We assume that values for A# and B# were assigned elsewhere in the program.

Sample Program

```
710 DEF FNV(T) = (1087 + SQR(273 + T))/16.52
720 INPUT "AIR TEMPERATURE IN DEGREES CELSIUS"; T
730 PRINT "THE SPEED OF SOUND IN AIR OF" T "DEGREES"
     " CELSIUS IS" FNV(T) "FEET PER SECOND."
```



DEFINT
Define Variables as Integers

DEFINT letter list
letter list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas. A letter-range is of the form:
letter1 – letter2

DEFINT causes variables beginning with any letter specified in *letter list* to be classified as integer, unless a type declaration character is added to the variable name. Integer values must be in the range [-32768,32767]. They are stored internally in two-byte, two's complement form.

DEFINT may be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore, it is normally placed at the beginning of a program.

Examples

```
DEFINT A, I, N
```

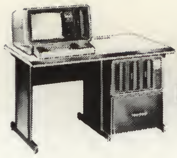
After the above line, all variables beginning with A, I, or N will be treated as integers. For example, A1, AA and I3 will be integer variables. However, A1#, AA# and I3# would still be double-precision variables, because type-declaration characters always override DEF statements.

```
DEFINT I-N
```

causes any variable beginning with the letters I through N to be treated as an integer variable.

Sample Program

```
880 DEFINT W
890 Z = 1.99999: W = 1.99999
900 PRINT "THE VALUE OF SINGLE-PRECISION Z IS" Z
910 PRINT "BUT THE VALUE OF INTEGER W IS" W
```



DEFSNG
Define Variables as Single-Precision

DEFSNG *letter-list*
letter-list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas. A letter-range is of the form:
a letter-range is of the form:
letter1 – letter2

DEFSNG causes variables beginning with any letter specified in *letter list* to be classified as single-precision, unless a type declaration character is added to the variable name. Single-precision values include 7 digits of precision, though only 6 are printed out.

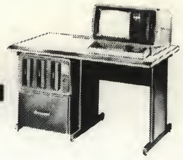
Example

```
DEFSNG I, W-Z
```

causes any variables beginning with the letters I or W through Z to be treated as single-precision. However, I% would still be an integer variable, and I# a double-precision variable, because of their type declaration characters.

Sample Program

```
960 CLS: DEFINT P: PI = 3.14159
970 PRINT "ALL P'S ARE INTEGERS: WE CAN ONLY MAKE PI =" PI
980 INPUT "WANT TO MAKE P'S SINGLE-PRECISION WITH DEFSNG (Y/
N)"; A$
990 IF A$ = "N" THEN END
1000 CLS: DEFSNG P: PI = 3.14159
1010 PRINT "NOW ALL P'S ARE SINGLE-PRECISION; WE CAN MAKE PI
 =" PI
```



DEFSTR
Define Variables as Strings

DEFSTR *letter-list*

letter-list is a sequence of individual letters or letter-ranges; the elements in the list must be separated by commas. A letter-range is of the form:

letter1 – letter2

DEFSTR causes variables beginning with any letter specified in *letter-list* to be classified as strings, unless a type declaration character is added to the variable name.

Example

```
DEFSTR C, L-Z
```

causes any variables beginning with the letters C or L through Z to be string variables, unless a type declaration character is added. After this line is executed, L1 = "WASHINGTON" will be valid.

Sample Program

```
70 S = 555: PRINT "S =" S
80 DEFSTR S
90 S = "SALTON SEA": PRINT "S = " S
```

DEFUSR
Define Point of Entry for USR Routine

DEFUSR*n* = *address*
n equals one of the digits 0,1,...,9; if *n* is omitted, 0 is assumed.
address specifies the entry address to a machine-language routine.
address must be in the range [−32768,32767].

DEFUSR lets you define the entry points for up to 10 machine-language routines.

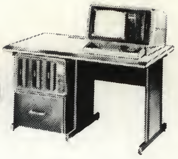
Examples

```
DEFUSR3 = &H7D00
```

assigns the entry point X'7D00', 32000 decimal, to the USR3 call. When your program calls USR3, control will branch to your subroutine beginning at X'7D00'.

```
DEFUSR = (BASE + 16)
```

assigns start address (BASE + 16) to the USR0 routine.



DIM
Set Up Array

DIM *array1 (dimension list), array2(dimension list), . . .*
*array1, array2, . . .*are variables which name the array(s).
dimension lists are of the form:
subscript1, subscript2,...
each *subscript* is a numeric expression specifying the highest-numbered element in that dimension of the array.
Note: The lowest element in a dimension is always zero.

This statement sets up one or more arrays for structured data processing. Each array has one or more dimensions.

Arrays may be of any type: string, integer, single-precision or double-precision, depending on the type of variable name used to name the array.

When the array is created, BASIC reserves space in memory for each element of the array. (For string arrays, BASIC reserves space for pointers to the string elements, not for the elements themselves.) All elements in a newly created array are set to zero (numeric arrays) or the null string (string arrays).

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a BASIC statement, e.g.,

```
A(5) = 300
```

If this is the first reference to array A( ), then BASIC will create the array and assign element A(5) the value of 300. Each dimension of an implicitly defined array is defined to be 11 elements deep, subscripts 0-10.

When an array has been defined, it cannot be re-dimensioned. You must clear the array first (with ERASE, CLEAR or NEW or other variable-clearing operation).

Examples

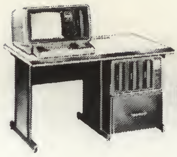
```
DIM AR(100)
```

Sets up a one-dimensional array AR( ), containing 101 elements: A(0), A(1), A(2), . . ., A(98), A(99), and A(100). The type of the array depends on the type of the name AR. Unless previously changed by a DEFINT, DEFDBL or DEFSTR statement, AR is a single-precision variable.

Note: The array AR( ) is completely independent of the variable AR.

```
DIM L1%(8,25)
```

Sets up a two-dimensional array L1%(,), containing 9 × 26 integer elements, L1%(0,0), L1%(1,0), L1%(2,0), . . ., L1%(8,0), L1%(0,1), L1%(1,1), . . ., L1%(8,1), . . ., L1%(0,25),L1%(1,25), . . ., L1%(8,25).



Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

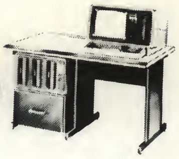
0,0	0,1	0,2	0,3	. . .	0,23	0,24	0,25
1,0	1,1	1,2	1,3	. . .	1,23	1,24	1,25
.							
.							
.							
7,0	7,1	7,2	7,3	. . .	7,23	7,24	7,25
8,0	8,1	8,2	8,3	. . .	8,23	8,24	8,25

```
DIM B1(2,5,8), CR(2,5,8), LY$(50,2)
```

Sets up three arrays:
B1 (, ,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.
LY (,) is two-dimensional, containing 51*3 string elements.

Sample Program

```
170 CLEAR 4000: CLS
180 INPUT "HOW MANY MEMBERS IN THE CLUB"; M
190 DIM L$(M,4)
200 FOR I = 1 TO M
210      PRINT "NAME OF MEMBER #" I;: LINE INPUT"? "; L$(I,1)
220      INPUT "AGE"; L$(I,2)
230      INPUT "PHONE"; L$(I,3)
240      LINE INPUT "ADDRESS? "; L$(I,4)
250 NEXT I
260 PRINT
270 PRINT "THE LIST IS STORED AS FOLLOWS:"
280 PRINT "NAME", "AGE", "PHONE", "ADDRESS"
290 PRINT STRING$(80, "-")
300 FOR I = 1 TO M
310      FOR J = 1 TO 4
320           PRINT L$(I,J),
330      NEXT J
340      PRINT
350 NEXT I
```



ERASE
Delete Array

ERASE *array1, array2,...*
array1, array2 are variable names for currently defined arrays.

The ERASE statement eliminates arrays from a program and allows their space in memory to be used for other purposes. ERASE will only operate on arrays. It can't be used to delete single elements of an array.

If one of the arguments of ERASE is a variable name which is not used in the program, an Illegal Function Call will occur.

Arrays deleted in an ERASE statement may be re-dimensioned.

Example

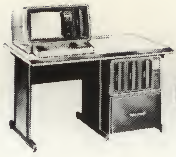
```
ERASE C,F, TABLE
```

Erases the three specified arrays.

Sample Program

```
400 DIM A(5,5):X=0
410      FOR I=0 TO 5
420            FOR J=0 TO 5
430            X = X + 1
440            A(I,J) = X
445            PRINT A(I,J),
450            NEXT J
460      NEXT I
470 ERASE A
480 DIM A(100)
```

The array that is set up in line 400 is destroyed by the ERASE A statement in line 470. The memory space which is thereby released is now available for further use. The array may be re-dimensioned, as we've chosen to do in line 480.



RANDOM
Reseed Random Number Generator

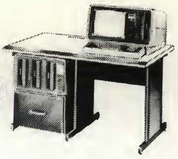
RANDOM

RANDOM reseeds the random number generator. If your program uses the RND function, the same sequence of pseudorandom numbers will be generated every time the Computer is turned on and the program loaded. Therefore, you may want to put RANDOM at the beginning of the program. This will help ensure that you get a different sequence of pseudorandom numbers each time you run the program.

Random needs to execute just once.

Sample Program

```
600 CLS: RANDOM
610 INPUT "PICK A NUMBER BETWEEN 1 AND 5"; A
620 B = RND(5)
630 IF A = B THEN 650
640 PRINT "YOU LOSE, THE ANSWER IS" B "-- TRY AGAIN."
645 GOTO 610
650 PRINT "YOU PICKED THE RIGHT NUMBER -- YOU WIN!": GOTO 610
```



REM
Comment Line (Remarks)

REM

REM instructs the Computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you look at a listing of your program, or someone else does, it will be easier to figure out.

If REM is used in a multi-statement program line, it must be the last statement.

An apostrophe (') may be used as an abbreviation for :REM.

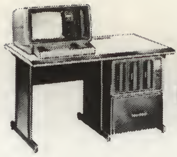
Example

```
' THIS IS A REMARK
```

Sample Program

```
2000 INPUT A              :REM  Input single-precision
2100 A = A/2              :REM  Find smaller values
2200 PRINT A              :REM  Print
2300 GOTO 2100            :REM  Loop for next
```

The above program shows some of the graphic possibilities of REM statements. Any alphanumeric character may be included in a REM statement, and the maximum length is the same as that of other statements: 255 characters total.



RESTORE
Reset Data Pointer

RESTORE

RESTORE causes the next READ statement to be executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.

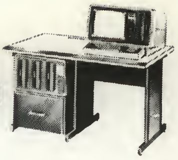
Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA THIS IS THE FIRST ITEM, AND THIS IS THE SECOND
```

When this program is run,

```
THIS IS THE FIRST ITEM    THIS IS THE FIRST ITEM
```

will be printed on the Display. Because of the RESTORE statement in line 170, the second READ statement starts over with the first DATA item.



Assignment

An assignment statement puts a certain value into a variable or field or trades the value of one variable with another.

```
CLR$ = "VERMILION"
```

This statement assigns the value VERMILION to CLR$

```
SWAP A%, B%
```

A% and B% exchange values with one another.

Keywords	Purpose
LET	Assign value to variable
LSET	Left-set data in direct access disk buffer
MID$=	Replace mid-string
READ	Get value from DATA statement
RSET	Right-set data in direct access disk buffer
SWAP	Exchange values of variables

LET
Assign Value to Variable

LET *variable* = *expression*

LET may be used when assigning values to variables. Model II BASIC doesn't require assignment statements to begin with LET, but you might want to use it to ensure compatibility with those versions of BASIC that do require it.

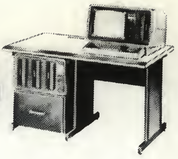
Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P = " P
```



LSET and RSET
Place Data in a Direct Access Buffer Field

LSET *name* = *data*
RSET *name* = *data*
name is a field (a string expression) name
data is the data (a string expression) to be placed in the buffer field.

These two statements let you place string data into fields previously set up by a FIELD statement.

Examples

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

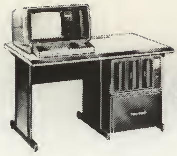
```
LSET NM$ = "JIM CRICKET,JR."
LSET AD$ = "2000 EAST PECAN ST."
```

put the data in the buffer as follows:

JIMƀCRICKET,JR.ƀƀƀ   2000ƀEASTƀPECANƀST.ƀƀƀƀƀƀ

Notice that filler blanks were placed to the right of the data strings in both cases. If we had used RSET statements instead of LSET, the filler spaces would have been placed to the left. This is the only difference between LSET and RSET.

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored.



MID$=
Replace Portion of String

MID$ (*oldstring, position, length*) = *replacement-string*
oldstring is the variable-name of the string you want to change
position is the numeric expression specifying the position of the first character to be changed
length is a numeric expression specifying the number of characters to be replaced
replacement-string is a string expression to replace the specified portion of *oldstring*

Note: If *replacement-string* is shorter than *length,* then the entire *replacement-string* will be used.

This statement lets you replace any part of a string with a specified new string, giving you a powerful string editing capability.

Note that the length of the resultant string is always the same as the original string.

Examples:

A$ = "LINCOLN" in the examples below:

```
MID$(A$, 3, 4) = "12345": PRINT A$
```

which returns LI1234N.

```
MID$(A$, 1, 2) = "": PRINT A$
```

which returns LINCOLN.

```
MID$(A$, 5) = "12345": PRINT A$
```

returns LINC123.

```
MID$(A$, 5) = "01": PRINT A$
```

returns LINC01N.

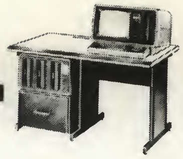
```
MID$(A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

Sample Program

```
770 CLS: PRINT: PRINT
780 LINE INPUT "TYPE IN A MONTH AND DAY MM/DD. "; S$
790 P = INSTR(S$, "/")
800 IF P = 0 THEN 780
810 MID$(S$, P, 1) = CHR$(45)
820 PRINT S$ " IS EASIER TO READ, ISN'T IT?"
```

This program uses INSTR to search for the slash ("/"). When it finds it (if it finds it), it uses MID$= to substitute a "—" (CHR$(45)) for it.



READ
Get Value from DATA Statement

READ *variable, . . .*

READ instructs the Computer to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement, etc. (An Out of Data error occurs if there are more attempts to READ than there are DATA items.)

Examples

```
READ T
```

reads a numeric value from a DATA statement.

```
READ S$, T, U
```

reads values for S$, T and U from a DATA statement.

Sample Program

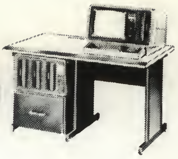
This program illustrates a common application for READ and DATA statements.

```
40 PRINT "NAME","AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$,AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERSON, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, END
```

RSET
Place Data in a Direct Access Buffer Field

RSET *name* = *data*
name is a field name
data is the data (a string expression) to be placed in the buffer field.

See LSET for details.



SWAP
Exchange Values of Variables

SWAP *variable1, variable 2*

The SWAP statement allows the values of two variables to be exchanged. Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not had values assigned to them, an Illegal Function Call error will result. Both variables must be of the same type or a Type Mismatch error will result.

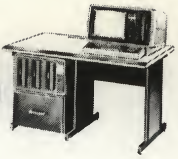
Example

```
SWAP F1#, F2#
```

The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
100 'BUBBLE SORT USING SWAP
110 DEFINT A-Z: DIM A(50)
120 A(0)=0
125 PRINT "HERE ARE 50 NUMBERS BETWEEN 1 AND 100"
130 FOR I=1 TO 50: A(I)=RND(100): PRINT A(I);: NEXT
170 PRINT:PRINT:
    PRINT"NOW SORTING DATA. START TIME = " TAB(40) TIME$
180 F=0: K=0     'F is set when a SWAP is made, K is counter
190 IF A(K)>A(K+1) THEN SWAP A(K), A(K+1): F=1  'SWAP & set F
200 K=K+1: IF K<50 THEN 190
210 IF F=1 THEN 180      'go through data again until F=0
220 PRINT: PRINT"DATA SORTED. END TIME = " TAB(40) TIME$
230 PRINT: PRINT"HERE IT IS IN ORDER:"
240 FOR I= 1 TO 50: PRINTA(I);: NEXT
```



Program Sequence

Control in a BASIC program normally proceeds from one line to the next higher-numbered line to the next higher-numbered line, until the end of the program is reached. The program sequence statements can be used to alter this step-by-step process. With the help of these statements, you can alter the transfer of control in your BASIC program to produce jumps to other parts of the program, iterative loops, and other useful control structures.

For example, the statement

```
IF NOT X > 5 AND NOT Y > 8 THEN 100
```

transfers control to line 100 if X is not greater than 5, and, at the same time, Y is not greater than 8.

```
FOR I = 1 TO 10000: NEXT I
```

Program control will pass back and forth between the FOR statement and the NEXT statement ten thousand times before moving on to the next line, causing a delay of approximately eleven seconds.

Keyword	Purpose
END	End program
FOR/NEXT	Set up loop
GOSUB	Call subroutine
GOTO	Branch to line number
IF...THEN...ELSE	Test conditional expression
ON . . .GOSUB	Multi-way subroutine call
ON . . .GOTO	Multi-way branch to line numbers
RETURN	Return from subroutine

END
Terminate Program

```
END
```

END terminates execution of a program. Some versions of BASIC require END as the last statement in a program. In Model II BASIC it is optional. END is primarily used in Model II BASIC to force execution to terminate at some point other than the last sequential line in the program.

Sample Program

```
40 INPUT S1,S2
50 GOSUB 100
55 PRINT H
60 END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

The END statement in line 60 prevents program control from "crashing" into the subroutine. Now line 100 can only be accessed by a branching statement such as line 50.

FOR/NEXT
Establish Program Loop

FOR *variable = initial value* TO *final value* STEP *increment*
NEXT *variable*

variable is any integer or single-precision variable name; *variable* is optional after NEXT.

initial value, final value, and *increment* are numeric constants, variables, or expressions.

STEP *increment* is optional; if STEP *increment* is omitted, a value of 1 is assumed.

FOR . . . TO . . . STEP/NEXT opens an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times.

The first time the FOR statement is executed, *variable* is set to *initial value.* Execution proceeds until a NEXT is encountered. At this point, *variable* is incremented by the amount specified in step *increment.* (If *increment* has a negative value, then *variable* is actually decremented.) STEP *increment* is often omitted, in which case an increment of 1 is used.

Then *variable* is compared with *final value.* If *variable* is greater than *final value,* the loop is completed and execution continues with the statement following NEXT. (If *increment* is a negative number, the loop ends when *variable* is **less** than *final value*.) If *variable* has not yet exceeded *final value*, control passes to the statement following the FOR statement.

Sample Programs

```
830 FOR I = 10 TO 1 STEP -1
840 PRINT I;
850 NEXT I
```

When this program is run, the following output is produced:

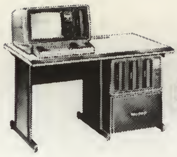
```
10  9  8  7  6  5  4  3  2  1
```

FOR/NEXT loops may be "nested":

```
880 FOR I = 1 TO 3
890 PRINT "OUTER LOOP"
900      FOR J = 1 TO 2
910      PRINT "     INNER LOOP"
920      NEXT J
930 NEXT I
```

NEXT can be used to close nested loops, by listing the counter-variables. For example, delete line 920 and change 930 to:

```
NEXT J, I
```



GOSUB
Go to Specified Subroutine

GOSUB *line number*

GOSUB transfers program control to the subroutine beginning at the specified line number. When the Computer encounters a RETURN statement in the subroutine, it then returns control to the statement which follows GOSUB. GOSUB is similar to GOTO in that it may be preceded by a test statement. **Every subroutine must end with a RETURN.**

Example

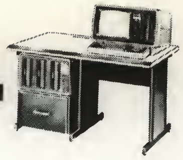
```
    GOSUB 1000
```

When this line is executed, control will automatically branch to the subroutine at 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

Control is transferred from line 260 to the subroutine beginning at line 280. Line 290 instructs the Computer to return to the statement immediately following GOSUB.



GOTO
Go To Specified Line Number

GOTO *line number*

GOTO transfers program control to the specified line number. Used alone, GOTO *line number* results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. GOTO *line number* causes execution to begin at the specified line number, *without an automatic* CLEAR. This lets you pass values assigned in the command mode to variables in the execute mode.

Example

```
GOTO 100
```

When this line is executed, control will automatically be transferred to line 100.

Sample Program

```
160 GOTO 200
170 PRINT "AND ARAMIS -- AND D'ARTAGNAN MAKES FOUR.": END
180 PRINT "PORTHOS, ";
190 GOTO 170
200 PRINT "ATHOS, ";
210 GOTO 180
```

IF . . . THEN . . . ELSE
Test Conditional Expression

IF *test* THEN *statement* or *line number* ELSE *statement* or *line number*
ELSE *statement* or *line number* is optional.

The IF . . . THEN . . . ELSE statement instructs the Computer to test the following logical or relational expression. If the expression is true, control will proceed to the THEN clause immediately following the expression. If the expression is false, control will jump to the matching ELSE statement (if one is included) or down to the next program line.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" : END
```

If X is greater than 127, control will pass to PRINT and then to END. If X is not greater than 127, control will jump down to the next line in the program, skipping the PRINT and END statements.

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

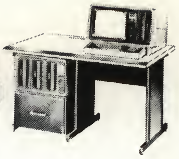
If both expressions are true, then Y will be assigned the value X + 180. Otherwise control will pass directly to the next program line, skipping the THEN clause.

```
IF A < B PRINT "A < B" ELSE PRINT "B < = A"
```

If A is less than B, the Computer prints the fact and then proceeds down to the next program line, skipping the ELSE statement. If A is not less than B, the Computer jumps directly to the ELSE statement and prints the specified message. **Then** control passes to the next statement in the program.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN 400 ELSE 370
```

If A$ is YES then the program branches to line 210. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO then the program branches to line 400. If A$ is any value besides NO or YES, the program skips to the second ELSE and the program branches to line 370.



```
IF A > .001 THEN B = 1/A: A = A/5: ELSE 1510
```

If the value of A is indeed greater than .001, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A is less than or equal to .001, then the program jumps directly over to ELSE, which then instructs it to branch to 1510. Note that GOTO is not required after ELSE.

Sample Program

IF THEN ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs.

```
1040 INPUT "ENTER TWO NUMBERS"; A, B
1050 IF A <= B THEN IF A < B THEN PRINT A; ELSE
PRINT "NEITHER "; ELSE PRINT B;
1060 PRINT "IS SMALLER THAN THE OTHER."
```

For any pair of numbers that you enter, this program will pick out and print the smaller of the two.

ON . . . GOSUB
Test and Branch to Subroutine

On *test-value* GOSUB *line number, line number, . . .*
test-value is a numeric expression between 0 and 255.

ON . . . GOSUB is a multi-way branching statement like ON GOTO, except that control passes to a subroutine rather than just being shifted to another part of the program. For further information, see ON GOTO.

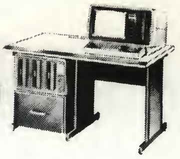
Example

```
ON Y GOSUB 1000, 2000, 3000
```

When program execution reaches the line above, if Y = 1, the subroutine beginning at 1000 will be called. If Y = 2, the subroutine at 2000 will be called. If Y = 3, the subroutine at 3000 will be called.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```



ON . . . GOTO
Test and Branch to Different Program Line

ON *test-value* GOTO *line number, line number, . . .*
test-value is a numeric expression between 0 and 255.

ON . . . GOTO is a multi-way branching statement that is controlled by a test value.

When ON . . .GOTO is executed, *test-value* is evaluated and the integer portion is obtained. We'll refer to this integer portion as J. The Computer counts over to the Jth line number in the list of line numbers after GOTO, and branches to this line number. If there is no Jth line number, then control passes to the next statement in the program.

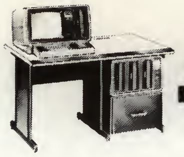
Notice that if *test-value* is less than zero, an error will occur. There may be any number of line numbers after GOTO.

Examples

```
ON MI GOTO 150, 160, 170, 150, 180
```

says "Evaluate MI.
If integer portion of MI equals 1 then go to line 150;
If it equals 2, then go to 160;
If it equals 3, then go to 170;
If it equals 4, then go to 150;
If it equals 5, then go to 180;

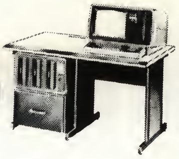
If the integer portion of MI doesn't equal any of the numbers 1 through 5, advance to the next statement in the program."



Sample Program

```
750 INPUT "TYPE IN ANY NUMBER" ; X
760 ON SGN(X) + 2 GOTO 770, 780, 790
770 PRINT "NEGATIVE": END
780 PRINT "ZERO" :END
790 PRINT "POSITIVE" :END
```

SGN(X) returns −1 for X less than zero; 0 for X equal to zero; and +1 for X greater than 0. By adding 2, the expression takes on the values 1, 2, and 3, depending on whether X is negative, zero, or positive. Control then branches to the appropriate line number.



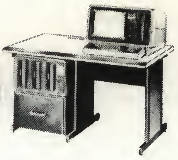
RETURN
Return Control to Calling Program

```
RETURN
```

RETURN ends a subroutine by returning control to the statement immediately following the most-recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS" ; R
350 GOSUB 370
360 PRINT "AREA IS"; A: END
370 A = 3.14 * R * R
380 RETURN
```



Input/Output

These statements perform input/output to the keyboard, video display, line printer, and disk files. They are grouped accordingly in this section.

Note: Before attempting any input/output to BASIC data files, you should read Chapter 4, **File Access Techniques**, and try out the sample programs given there.

Keyword	Purpose
INPUT	Input value from keyboard
LINE INPUT	Input line from keyboard
CLS	Clear video display
PRINT	Print to video display
LPRINT	Print to line printer
CLOSE	Close access to a disk file
FIELD	Organize a disk buffer (direct access)
GET	Get a record from a disk file (direct access)
INPUT#	Read from a disk file (sequential access)
LINE INPUT#	Read a line from a disk file (sequential access)
OPEN	Open a disk file (direct or sequential access)
PRINT#	Write to a disk file (sequential access)
PUT	Put a record into a disk file (direct access)

INPUT
Input Data to Program

```
INPUT "message"; variable 1, variable 2, . . .
```

When BASIC encounters the INPUT statement in a program it stops execution of the program until you enter certain values from the keyboard. The INPUT statement may specify a list of string or numeric variables, indicating string or numeric values to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, *in that order.*

When the statement is encountered, the Computer will display a ?. You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

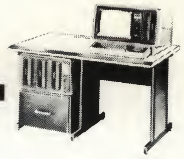
If you ENTER the values one at a time, the Computer will display a ??, indicating that more data is expected. Continue entering data until all the variables have been set, at which time the Computer will advance to the next statement in your program.

Be sure to enter the correct type of value according to what is called for by the INPUT statement. For example, you can't input a string-value into a numeric variable. If you try, the Computer will display a ?REDO FROM START and give you another chance to enter the correct type of data value, starting with the *first* value to be called for by the INPUT list.

If you ENTER more data elements than the INPUT statement specifies, the Computer will display the message ?EXTRA IGNORED and continue with normal execution of your program.

You can include a "prompting message" in your INPUT statement. This will make it easier to input the data correctly. The prompting message must immediately follow INPUT. It must be enclosed in quotes, and it must be followed by a semicolon.

You can enter any valid constant. 2, 105, 1, 3#, etc. are all valid constants.



Examples

```
INPUT Y%
```

If this line were part of your program, when this line is reached, you must type any number and press ENTER before the program will continue.

```
INPUT SENTENCE$
```

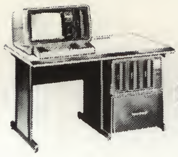
Here you would have to type in a string when this line is reached. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME, AGE)"; N$, A
```

This line would print a message on the screen which would help the person at the keyboard to enter the right sort of data.

Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT" CINT(X * .38) "POUNDS."
```



LINE INPUT
Input a Line from Keyboard

LINE INPUT [*"prompt"*] ;*variable*
prompt is a prompting message
variable is the name that will be assigned to the line you type in

LINE INPUT (or LINEINPUT – the space is optional) is similar to INPUT, except:

- The Computer will not display a question mark when waiting for your operator's input
- Each LINE INPUT statement can assign a value to just one variable
- Commas and quotes your operator can use as part of the string input
- Leading blanks are not ignored – they become part of *variable*
- The only way to terminate the string input is to press **ENTER**

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, colons, etc.). The **ENTER** key serves as the only delimiter. If you want anyone to be able to input information into your program without special instructions, use the LINE INPUT statement.

Some situations require that you input commas, quotes and leading blanks as part of the data. LINE INPUT serves well in such cases.

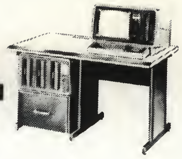
Examples:

```
LINE INPUT A$
```

Input A$ without displaying any prompt.

```
LINE INPUT "LAST NAME, FIRST NAME? ";N$
```

Displays a prompt message and inputs data. Commas will not terminate the input string, as they would in an input statement.



Sample Program

```
200 REM   CUSTOMER SURVEY
205 CLEAR 1000
207 PRINT
210 LINE INPUT "TYPE IN YOUR NAME "; A$
220 LINE INPUT "DO YOU LIKE YOUR COMPUTER? "; B$
230 LINE INPUT "WHY? "; C$
235 PRINT
240 PRINT A$ : PRINT
250 IF B$= "NO" THEN 270
260 PRINT "I LIKE MY COMPUTER BECAUSE "; C$ :END
270 PRINT "I DO NOT LIKE MY COMPUTER BECAUSE "; C$
```

Notice that when line 210 is executed, a question mark is not displayed after the statement, "Type in your name". Also, notice on line 230 you can answer the question "Why" with a statement full of delimiters, commas and quotes.

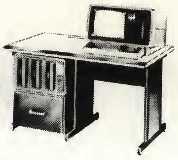
CLS
Clear Screen

```
CLS
```

CLS clear the screen. It fills the Display with blanks and moves the cursor to the upper-left corner. Alphanumeric characters are wiped out as well as graphics blocks. CLS can be very useful if you should want to present an attractive Display output.

Sample Program

```
540 CLS
550 FOR I = 1 TO 24
560 PRINT STRING$(79,33)
570 NEXT I
580 GOTO 540
```



PRINT, PRINT@, PRINT TAB, PRINT USING
Output to Display

PRINT@ *position, item list*
  @ *position* is a number between 0 and 1919. or
  @ *position* is two numbers, (*row, column*), *row* between 0 and 23 and *column* between 0 and 79. If @ *position* is omitted, the current cursor position is used.
  item list is a list composed of any of the following items:
    TAB (*number*)
      number is a numeric expression between 0 and 255
    expressions,
  where any of these items may be separated by the optional delimiters "," and";".

PRINT@ *position,* USING *format; item list*
  format is one or more of the field specifiers #, *, $, \ , !, " " (space), or any alphanumeric character.
  item list is a list composed of string or numeric expressions, which must be separated by the delimiters "," or ";".

PRINT prints an item or a list of items on the Display. The items to be printed may be separated by commas or semicolons.

If commas are used, the cursor automatically advances to the next tab position before printing the next item. If semicolons are used, spaces are not inserted between the items printed on the Display. There are five tab positions to a line, at columns 0, 14, 28, 42 and 56.

Use semi-colon as delimiter between items to avoid ambiguities. For example, a semi-colon is required in A;B but not in A$B$.

A semicolon or comma at the end of a line overrides the cursor-return so that the next PRINT begins where the last one left off. If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

Positive numbers are printed with a leading blank, instead of a plus sign. All numbers are printed with a trailing blank. No blanks are inserted before or after strings; you can insert them with the help of quotation marks.

Examples

```
        PRINT "I" ,, "VOTED" ; "FOR" ; "THAT" ; "RASCAL" ,
"DEWEY"
```

"I" is printed at tab position 0. "VOTEDFORTHATRASCAL" is printed at 28; "DEWEY" at 56.

```
    PRINT A$ B$ C$
```

This line is fully equivalent to

```
    PRINT A$;B$;C$
```

Sample Program

```
    70 N = 6
    80 A$ = "EDSELS": B$ = " AND MY WIFE OWNS 8."
    90 PRINT "I OWN"; N; A$;
    100 PRINT B$
```

When run, this program gives

```
     I OWN 6 EDSELS AND MY WIFE OWNS 8.
```

PRINT @ *n*,
PRINT @ (*row, column*),

PRINT@ specifies exactly where printing is to begin. The location specified must be a number between 0 and 1919, or a pair of numbers (**r**, **c**) with $0 <= \mathbf{r} <=$ 79 and $0 <= \mathbf{c} < 24$.

Whenever you cause something to PRINT@ on the bottom line of the Display, there is an automatic line feed; everything on the Display moves up one line. To suppress this automatic line feed, use a trailing semicolon at the end of the statement.

Examples

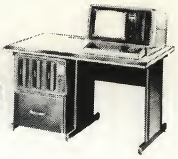
```
    PRINT @ (11,39), "*"
```

Prints an asterisk in the middle of the Display.

```
    PRINT @ 0, "*"
```

Prints an asterisk at the top left corner of the Display.

See "Using the Video Display" in Chapter 1 for an illustration of the print positions.



```
PRINT@ 550, "LOCATION 550"
```

Run this to find out where position 550 is.

```
PRINT@ 1000, X
```

Let's say the value of X in the above example is 7. "7" will be printed at location 1001, not 1000. Recall that a positive number will be printed with a leading blank to indicate its sign rather than a plus sign. So a space is printed at 1000 and the number itself is printed at 1001.

Sample Program

```
150 LINE INPUT "TYPE SOMETHING IN.  YOU'LL GET AN ECHO."; L$
155 CLS
160 PRINT@ 500, L$
170 PRINT@ 1000, L$
180 PRINT@ 1500, L$
```

PRINT TAB (*n*)

PRINT TAB moves the cursor to the specified position on the current line (or on succeeding lines if you specify TAB positions greater than 79). TAB may be used more than once in a print list.

Since numerical expressions may be used to specify a TAB position, TAB can be very useful in creating tables, graphs of mathematical functions, etc.

TAB can't be used to move the cursor to the left. If the cursor is to the right of the specified position, the TAB statement will simply be ignored.

Example

```
PRINT TAB(5) "TABBED 5"; TAB(25) "TABBED 25"
```

Notice that no punctuation is needed after the TAB modifiers.

Sample Program

```
220 CLS
230 PRINT TAB(2) "CATALOG NO."; TAB(16) "DESCRIPTION OF ITEM";
240 PRINT TAB(39) "QUANTITY"; TAB(51) "PRICE PER ITEM";
245 PRINT TAB(69) "TOTAL PRICE"
```

PRINT USING *format*

The PRINT USING statement allows you to specify a format for printing string and numeric values. It can be used in applications such as printing report headings, accounting reports, checks, or wherever a specific print format is required.

The PRINT USING statement ordinarily takes this form:

PRINT USING *format; item list*

PRINT USING takes the value *item list,* inserts it into the expression *format* as directed by the field specifiers of *format,* and prints the resulting expression. *format* may be expressed as a variable as well as a constant.

Note: PRINT USING does not automatically print leading and trailing blanks around numbers, except as indicated in *format.*

Examples of Field Specifiers

In all the examples below, the first line represents a program line as you might type it in; the second line is the value returned after the first line has been run.

The following field specifiers may be used as part of *format*:

This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros. If the numeric field is too small to hold a particular number, the number will be displayed with a leading % sign.

```
PRINT USING "#####"; 66.2
 66
```

. The decimal point can be placed anywhere in the numeric field established by the # sign. Rounding-off will take place when digits to the right of the decimal point are suppressed.

```
PRINT USING "##.#"; 58.76
58.8
```

, The comma — when placed in any position between the first digit and the decimal point — will display a comma to the left of every third digit as required. The comma establishes an additional position in the field.

```
PRINT USING "##########,"; 123456789
123,456,789
```

****** Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.

```
PRINT USING "**####"; 44.0
****44
```

$$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, the dollar sign will occupy the first position preceding the number.

```
PRINT USING "$$##.##"; 118.6735
$118.67
```

**$ If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the * sign and the $ sign will again position itself in the first position preceding the number.

```
PRINT USING "**$#.##"; 8.333
**$8.33
```

+ When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers or as a − for negative numbers.

```
PRINT USING "+**######"; 75200
**+75200

PRINT USING "+###"; -216
-216
```

− When a − sign is placed at the end of the field, it will cause a negative sign to appear after all negative numbers. A space will appear after positive numbers.

```
PRINT USING "####.#-"; -8124.420
8124.4-
```

! This causes the Computer to use the first string character of the current value.

```
PRINT USING "!"; "TANZANIA"
T
```

\ *spaces* \ To specify a string field of more than one character, \spaces\\ is used. The length of the field will be the number of spaces between the \\ \\ signs plus 2. Use **CTRL 9** to enter the backslash character.

One space between the backslashes:

```
PRINT USING "\ \"; "TANZANIA"
TAN
```

Four spaces between the backslashes:

```
PRINT USING "\    \"; "TANZANIA", "ETHIOPIA"
TANZANETHIOP
```

Any other character that you include in *format* will be displayed as a string literal.

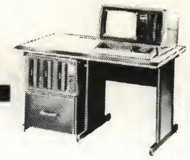
```
PRINT USING "$$.## BUCKS"; 8.625
$8.63 BUCKS
```

If *item list* is a numeric value, the % sign is automatically printed if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal will be displayed preceded by this sign.

```
PRINT USING "###.#"; 100000
%100000.0
```

^^^^ Indicates the number should be printed out in exponential (E or D) format.

```
PRINT USING "##.####^^^^"; 123456
 1.2346E+05
```



Sample Program

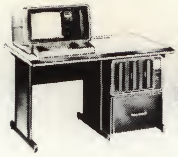
```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P
470 CLS: PRINT "PAY TO THE ORDER OF ";
480 PRINT USING "!! !! "; F$; "."; M$; ".";
490 PRINT L$
500 PRINT: PRINT USING A$; P
```

In line 480, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the two spaces in "!!b̸!!b̸". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for *format* and P for *item list* in line 500. Any serious use of the PRINT USING statement would probably require the use of variables at least for *item list* rather than constants. (We've used constants in our examples for the sake of better illustration.)

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,435.60 DOLLARS
```



LPRINT, LPRINT TAB, LPRINT USING
Output to Printer

LPRINT *item list*
  item list is a list composed of any of the following items:
    TAB (*number*)
      number is a numeric expression between 0 and 1919
    string or numeric expressions
  where any of these items may be separated by commas or semi-colons

LPRINT USING *format; item list*
  format is one or more of the field specifiers #, *, $,\ , !, or other characters.
  item list is a list composed of string or numeric expressions, which must be
    separated by commas or semi-colons.

LPRINT, LPRINT TAB, and LPRINT USING allow you to output to the Line Printer.

Examples

```
LPRINT (A * 2)/3
```

Sends the value of the expression (A * 2)/3 to the Line Printer.

```
LPRINT TAB(50) "TABBED 50"
```

Moves the Line Printer carriage to tab position 50 and prints TABBED 50.

```
LPRINT USING "######.#"; 2.17
```

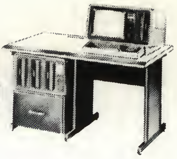
Sends the formatted value ƀƀƀƀ 2.2 to the Line Printer.

For more examples and a more detailed explanation of how to use these statements, see PRINT.

Sample Program

```
330 INPUT X
340 IF X<0 THEN 330
350 LPRINT "SQUARE ROOT IS" SQR(X)
360 END
```

Note: Before using LPRINT, you must initialize the printer software with the TRSDOS FORMS command.



CLOSE
Close Access to File

CLOSE *buffer-number, buffer-number, . . .*
buffer-number = 1,2,3, . . .,15
If *buffer-number* is omitted, all open files will be closed.

This command terminates access to a file through the specified buffer or buffers. If buffer-number has not been assigned in a previous OPEN statement, then

CLOSE *buffer-number*

has no effect.

Do not remove a diskette which contains an Open file. Close the file first. This is because the last records may not have been written to disk yet. Closing the file will write the data, if it hasn't already been written.

The following actions and conditions cause all files to be closed:

CLEAR or CLEAR n
NEW
RUN (except RUN *file*, R)
LOAD (except LOAD *file*, R)
MERGE
Editing a program line
Adding or deleting a program line

Examples

```
CLOSE 1,2,8
```

Terminates the file assignments to buffers 1, 2, and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST% + COUNT%
```

Terminates the file assignment to the buffer specified by the sum FIRST% + COUNT%.

FIELD
Organize a Direct File-Buffer Into Fields

FIELD *buffer-number, length* AS *name, length* AS *name, . . .*
buffer-number specifies a direct-access file buffer (1,2,3, . . .15)
length gives the number of bytes in the field
name defines a variable name for the field

The FIELD statement is used to organize a direct file buffer so data can be passed from BASIC to disk and disk to BASIC. Before fielding a buffer, you must use an OPEN statement to assign that buffer to a particular disk file. (The direct access mode, i.e., OPEN "D", . . . must be used.) The sum of all lengths should equal the record length assigned when the file was opened.

You may use the FIELD statement any number of times to "re-organize" a file buffer. FIELDing a buffer does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Furthermore, two or more field names can reference the same area of the buffer.

Examples

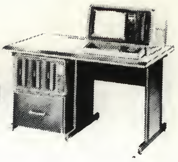
```
FIELD 1, 128 AS A$, 128 AS B$
```

This statement tells BASIC to assign two 128-byte fields to the variables A$ and B$. If you now print A$ or B$, you will see the contents of the field. Of course, this value would be meaningless unless you've previously used GET to read a 256-byte record from disk.

Note: All data – both strings and numbers – must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, and MKD$/CVD) for converting numbers to strings and strings to numbers.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS ST$,7 AS ZP$
```

The first 16 bytes of buffer 3 are assigned the field name NM$; the next 25 bytes, AD$; the next 10, CY$; the next 2, ST$; the next 7, ZP$.



GET
Get a Record from Disk (Direct Access)

GET *buffer-number, record number*
buffer-number specifies a direct access file buffer (1,2,3, . . .15)
record number specifies which record to GET in the file; if omitted, the current record will be read.

This statement gets a data record from a disk file and places it in the specified buffer. Before using GET, you must open the file and assign a buffer to it.

When BASIC encounters the GET statement, it reads the record number from the file and places it into the buffer. If you omit record number it will read the next record. The actual number of bytes read equals the record length, set when the file is Opened.

The next record is the record whose number is one greater than that of the last record accessed. The first time you access a file via a particular buffer, the current record is set to 1.

GET with a Default Record Number

The first time you use GET or PUT after opening a file, you must specify the record number. For subsequent GET or PUT statements, you can omit the record number, in which case BASIC will use the record following the last record accessed. In short:

1. For the first direct access of a file, use:

   GET *buffer-number, record-number*

2. For subsequent accesses, you may use

   GET *buffer-number*

   to get the next record (the record following the last record accessed).

Examples

```
GET 1
```

Gets the next record into buffer 1.

```
GET 1, 25
```

Gets record 25 into buffer 1.

INPUT#
Sequential Read from Disk

INPUT# *buffer-number, variable-list*
buffer-number specifies a sequential input file buffer (1,2,3, . . .15)
variable-list is a sequence of variable names to contain the data from the file

This statement inputs data from a disk file.

With INPUT#, data is input sequentially. That is, when the file is opened, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and re-open it.

INPUT# doesn't care how the data was placed on the disk – whether a single PRINT# statement put it there, or whether it required ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

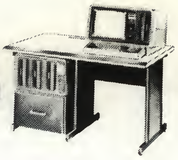
Examples

```
INPUT#1, A,B
```

Sequentially inputs two numeric data items from disk and places them in A and B. File-buffer #1 is used.

```
INPUT#4, A$, B$, C$
```

Sequentially inputs three string data items from disk and places them in A$, B$, and C$. File-buffer #4 is used.



LINE INPUT#
Read Line of Text from Disk

LINE INPUT# *buffer-number, name*
    buffer-number specifies a sequential input file buffer (1,2,3, . . .15)
    variable is the variable name to contain the string data

Similar to LINE INPUT from the keyboard, LINE INPUT# reads a "line" of string data into name. LINE INPUT# is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to

- a carriage return character which is not preceded by a line feed character
- the end-of-file
- the 255th data character (the 255th character is included in the string)

Other characters encountered – quotes, commas, leading blanks, line feed – carriage return sequences – are included in the string.

Example

If the data on disk looks like

```
10 CLEAR 500
20 OPEN "I", 1, "PROG"
```

then the statement

```
LINE INPUT#1,A$
```

could be used repetitively to read each program line, one line at a time.

OPEN
Open a Disk File

OPEN *mode, buffer-number, file, record-length*
mode is a string expression or constant of which only the first character is significant; this character specifies the mode in which the file is to be opened: I for sequential input, O for sequential output, D for direct-access input-output. "R" can also be used for direct ("random") I/O.
buffer-number specifies a buffer to be assigned the file specified by filespec
file defines a TRSDOS file specification
record-length = 0,1,2 . . .255. If record-length is omitted or if a value of 0 is used, the record length will be 256.

This statement makes it possible to access a file. *mode* determines what kind of access you'll have via the specified buffer. *buffer-number* determines which buffer will be assigned to the file. *file* names the file to be accessed. If *file* does not exist, then TRSDOS may or may not create it, depending on the access mode.

When a file is open, it is referenced by the buffer-number which was assigned to it. GET *buffer-number*, PUT *buffer-number,* PRINT# *buffer-number,* INPUT# *buffer-number,* all reference the file which was opened via *buffer-number*. The mode must be correct.

Once a buffer has been assigned to a file with the OPEN statement, that buffer can't be used in another OPEN statement. You have to Close it first.

Examples

```
OPEN "O", 1, "CLIENTS/TXT"
```

Opens the file CLIENTS/TXT for sequential output. Buffer 1 will be used. If the file does not exist, it will be created. If it already exists, then its previous contents are lost.

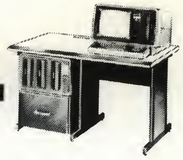
```
OPEN "D", 2, "DATA/BAS.SPECIAL"
```

Opens the file DATA/BAS with the password SPECIAL in the direct access mode. Buffer number 2 is used. If DATA/BAS does not exist, it will be created on the first non write-protected drive.

```
OPEN "D", 5, "TEST/BAS", 64
```

Opens the file TEXT/BAS for direct access. Buffer number 5 is used. The record length is 64. If this record-length does not match the record-length assigned to TEXT/BAS when the file was originally opened, an error will occur.

See Chapter 4 for programming information.



PRINT#
Sequential Write to Disk File

PRINT# *buffer-number, item list*
buffer-number specifies a sequential output file buffer (1,2,3, . . .15)
item list specifies the data to be written to disk
It is analogous to *item list* used in a normal PRINT statement. See PRINT.

This statement writes data sequentially to the specified file. When you first open a file for sequential output, a pointer is set to the beginning of the file. Thus the first PRINT# places data at the beginning of the file. At the end of each PRINT# operation the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the Display creates on the screen. Remember this, and you'll be able to set up your PRINT# list correctly for access by one of more INPUT statements.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

Examples

If A = 123.45

```
PRINT#1,A
```

will write a nine-byte character sequence onto disk:

b̸ 123.45 b̸   carriage return

where "b̸" indicates a blank.

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT to Display statements. For example, if A = 2300 and B = 1.303, then

```
PRINT#1, A,B
```

places the data on disk as

b̸ 2300 b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸ 1.303b̸   carriage return

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semicolons instead of commas.

Files can be written in a carefully controlled format using PRINT# USING. Or you can use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VAN", and C$ = "BEETHOVEN". Then the statement

```
PRINT#1,USING"!.!.\  \";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET
```

(In this case, we didn't want to add any explicit delimiters.) See PRINT for more information on the USING option.

PUT
Write a Record to Disk (Direct Access)

PUT *buffer-number, record number*
buffer-number specifies a direct access file buffer (1,2,3, . . . 15)
record number specifies the record number of the file. If record number is omitted, the current record number is used

This statement moves data from the buffer of a file into a specified place in the file. Before putting data into a file, you must

1. Open a file, which assigns a buffer and defines the access mode (which must be D)
2. Field the buffer, so you can
3. Place data into the buffer with LSET and RSET statements.

The first time you access a file via a particular buffer, the next record is set equal to 1. (The next record is the record whose number is one greater than the last record accessed.)

If *record number* is higher than the end-of-file record number, then *record number* becomes the new end-of-file record number.

PUT with a Default Record Number

The first time you use GET or PUT after opening a file, you must specify the record number. For subsequent GET or PUT statements, you can omit the record number, in which case BASIC will use the record following the last record accessed. In short:

1. For the first direct access of a file, use:

   PUT *buffer-number, record-number*

2. For subsequent accesses, you may use

   PUT *buffer-number*

   to put the next record (the record following the last record accessed).

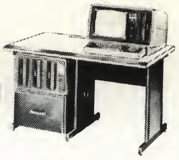
Examples

```
PUT 1
```

Puts the next record into buffer 1.

```
PUT 1,25
```

Puts record 25 into buffer 1.



Debug Statements

These statements can help you debug (isolate errors in) programs you are developing. They can also be used to create routines that trap errors without provoking BASIC error messages.

Keyword	Purpose
CONT	Continue execution of program
ERL	Get line number after error
ERR	Get error code after error
ERROR	Simulate error
ON ERROR GOTO	Set up error-trap
RESUME	End error trap
STOP	Stop execution
TROFF	Turn trace off
TRON	Turn trace on

CONT
Resume Execution of Program

```
CONT
```

When program execution has been stopped (by the BREAK key or by a STOP statement in the program), type CONT and ENTER to continue execution at the point where the stop or break occurred. During such a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT and ENTER and execution will continue with the current variable values. CONT, when used with STOP and the BREAK key, is primarily a debugging tool.

Note: You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.

See also STOP.

ERL
Get Line Number of Error

```
ERL
```

ERL returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number 0 is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, 65535 (the largest number representable in two bytes) is returned.

Examples

```
PRINT ERL
```

Prints the line number of the error.

```
E = ERL
```

Stores the error's line number for future use.

Sample Program

```
1999 REM ERL PROGRAM
2000 CLEAR 100: ON ERROR GOTO 2125
2010 INPUT"WHAT NAME ARE YOU LOOKING FOR";Q$
2020 READ T$
2030 IF Q$=T$ THEN PRINT T$" IS IN THE LIST": GOTO 2060
2040 GOTO 2020
2050 PRINT Q$" IS NOT IN THE LIST.": RESTORE: GOTO 2010
2060 INPUT"WHAT IS YOUR FAVORITE FRUIT";F$
2070 READ T$
2080 IF F$=T$ THEN PRINT"YOU'RE IN LUCK--WE HAVE SOME.":
     RESTORE: GOTO 2010
2090 GOTO 2070
2100 PRINT"SORRY--WE DON'T HAVE ANY "F$"S": RESTORE: GOTO 2060
2110 DATA TOM, DICK, HARRY, JAMES, ROBERT, SUE, SALLY,
     CERELLE, MARY
2120 DATA WATERMELON, PEACH, PEAR, ORANGE, APPLE, CHERRY,
     TOMATO, AVOCADO
2125 IF ERR<>4 THEN ON ERROR GOTO 0
2130 IF ERL=2020 THEN RESUME 2050
2140 IF ERL=2070 THEN RESUME 2100
2150 ON ERROR GOTO  0
```

ERR
Get Error Code

```
ERR
```

ERR is similar to ERL, except that ERR returns the code of the error rather than the line in which the error occurred. ERR is normally used inside an error-handling routine accessed by ON ERROR GOTO. See the section on error codes in the Appendix.

Examples

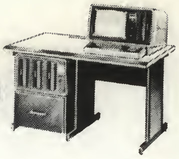
```
IF ERR = 7 THEN 1000 ELSE 2000
```

If the error is an Out of Memory error (code 7) the program branches to line 1000; if it is any other error, control will instead go to line 2000.

Sample Program

```
2160 ON ERROR GOTO 2220
2170 READ A
2180 PRINT A
2190 GOTO 2170
2200 PRINT "DATA HAS BEEN READ IN"
2210 END
2220 IF ERR = 4 THEN RESUME 2200
2230 ON ERROR GOTO 0
2232 DATA  4, 2, 0, 5, 9, 2, 2, 31, 7, 13, 1
```

This program "traps" the Out of Data error, since 4 is the code for that error.



ERROR
Simulate Error

ERROR *code*
code is a numeric expression in the range [0,255]

ERROR lets you simulate a specified error during program execution. The major use of this statement is for testing an ON ERROR GOTO routine. When the ERROR *code* statement is encountered, the Computer will proceed exactly as if that error had occurred. Refer to the Appendix for a listing of error codes and their meanings.

Example

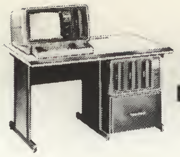
```
ERROR 1
```

When the program reaches this line, a Next Without For error (code 1) will "occur", and the Computer will print a message to this effect.

Sample Program

```
2240 INPUT N
2250 ERROR N
```

When you input one of the error code numbers, that error will be simulated in line 2250.



ON ERROR GOTO
Set Up Error-trapping Routine

ON ERROR GOTO *line number*

When the Computer encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine which will allow your program to "recover" from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement.

For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

The ON ERROR GOTO must be executed before the error occurs or it will have no effect. The ON ERROR GOTO statement can be disabled by executing the statement, ON ERROR GOTO 0. If you use this inside an error-trapping routine, BASIC will handle the current error normally. The error handling routine must be terminated by a RESUME statement. See RESUME.

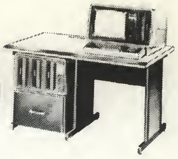
Examples

```
ON ERROR GOTO 1500
```

If an error occurs in your program anywhere after this line, control will branch to line 1500.

Sample Program

For the use of ON ERROR GOTO in a program, see the sample programs for ERL and ERR.



RESUME
Terminate Error-Trapping Routine

RESUME *line number*
line number is optional. If omitted, execution resumes at the beginning of the statement causing the error.

RESUME NEXT
Execution resumes **after** the statement causing the error.

RESUME terminates an error-handling routine by specifying where normal execution is to resume. Place a RESUME statement at the end of an error-trapping routine. That way later errors can also be trapped.

RESUME without an argument and RESUME 0 both cause the Computer to return to the statement in which the error occurred.

RESUME followed by a line number causes the Computer to branch to the specified line number.

RESUME NEXT causes the Computer to branch to the statement following the point at which the error occurred.

Examples

```
RESUME
```

If an error occurs, when program execution reaches the line above, control will be transferred to the statement in which the error occurred.

```
RESUME 10
```

If an error occurs, control will be transferred to line 10 after the problem has been fixed.

Sample Program

For the use of RESUME in a program, see the sample programs for ERL and ERR.

STOP
Interrupt Execution of Program

```
STOP
```

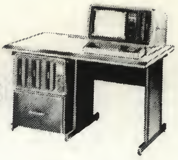
STOP interrupts the execution of your program and prints the words BREAK IN followed by the number of the line that contains the STOP. STOP is primarily a debugging aid. During the break in execution, you can examine variables or change their values.

The CONT command is used to resume execution at the point where it was halted. But if the program itself is altered during the break, CONT can't be used.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number between 1 and 10 will be assigned to X and program execution will halt at line 2270. You can now examine the value of X with PRINT X. Type CONT to start the cycle again.



TROFF, TRON
Turn Trace Function Off, On

```
TROFF
TRON
```

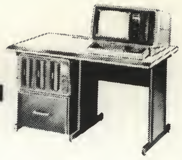
TRON turns on a trace function that lets you follow program flow for debugging and for analysis of the execution of the program. Each time the program advances to a new program line, that line number will be displayed inside a pair of brackets. TROFF turns trace off.

Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

The above three lines might be helpful in assuring you that line 2300 is actually being executed, since each time it is executed [2300] will be printed on the Display. (We assume the program doesn't jump directly to line 2300 without passing through line 2290, which would execute the assignment statement without turning the trace on.)

After a program is debugged, the TRON and TROFF statements can be removed.



Numeric Functions

All of these functions return a number. You can use them anywhere a numeric expression is called for. Notice that several string-related functions (ASC, INSTR, LEN, VAL) are included in this group.

Keyword	Purpose
ABS	Compute absolute value
ASC	Get ASCII code
ATN	Compute arctangent
CDBL	Convert to double-precision
CINT	Return largest integer not greater than argument
COS	Compute cosine
CSNG	Convert to single-precision
EXP	Compute natural exponential
FIX	Truncate to whole number
INSTR	Search for specified string
INT	Return largest whole number not greater than argument
LEN	Get length of string
LOG	Compute natural logarithm
RND	Return pseudorandom number
SGN	Get sign
SIN	Compute sine
SQR	Compute square root
TAN	Compute tangent
VAL	Evaluate string

ABS
Compute Absolute Value

ABS (*number*)
number is any numeric expression

ABS returns the absolute value of the argument, i.e., the magnitude of the number without respect to its sign. ABS(x)=x for x greater than or equal to zero, and ABS(x)=−x for x less than zero.

The result is always the same precision as the argument.

Examples

```
X = ABS(Y)
```

The absolute value of Y is assigned to X.

```
IF ABS(X) < 1E-6 THEN PRINT "TOO SMALL"
```

TOO SMALL is printed only if the absolute value of X is less than the indicated number.

Sample Program

```
100 INPUT "WHAT'S THE TEMPERATURE OUTSIDE (DEGREES F)"; TEMP
110 IF TEMP < 0 THEN PRINT "THAT'S" ABS(TEMP)
    "BELOW ZERO!  BRR!": END
120 IF TEMP = 0 THEN PRINT "ZERO DEGREES!  MITE COLD!": END
130 PRINT TEMP "DEGREES ABOVE ZERO?  BALMY!": END
```

ASC
Get ASCII Code

ASC (*string*)
string is a string expression. If the length of string equals zero, an Illegal Function Call will occur.

ASC returns the ASCII code of the first character of the string. The value is returned as a decimal number.

Examples

```
PRINT ASC("A")
PRINT ASC("AB")
```

Both lines will print 65, the ASCII code for "A".

```
PRINT ASC(RIGHT$(T$, 1))
```

Prints the ASCII code of the last character of T$.

Sample Programming

Refer to the ASCII code table in the Appendix. Note that the ASCII code for a lower-case letter is equal to that letter's upper case code plus 32. So ASC can be used to convert lower case to upper case, simply by subtracting 32 from ASC(x). For instance:

```
140 INPUT "LETTER (a-z)"; X$
150 IF X$ >= "a" AND X$ <= "z" THEN X$ = CHR$(ASC(X$) -32)
160 PRINT X$
```

ASC can be used to make sure that a program is receiving the proper input. Suppose you've written a program that requires the user to input hexadecimal digits 0-9, A-F. To make sure that only those characters are input, and exclude all other characters, you can insert the following routine.

```
100 INPUT"ENTER A HEXADECIMAL VALUE (0-9,A-F)";N$
110 A=ASC(N$)           'get ASCII code
120 IF A>47 AND A<58 OR A>64 AND A<71 THEN PRINT"OK.": GOTO 100
130 PRINT"VALUE NOT OK.": GOTO 100
```

ATN
Compute Arctangent

ATN (*number*)
number is a numeric expression

ATN returns the angle whose tangent is number. The angle will be in radians; to convert to degrees, multiply ATN(X) by 57.29578.

The result is always single-precision.

Examples

```
X = ATN(Y/3)
```

Assigns the value of the arctangent of Y/3 to X.

```
PRINT ATN(1.0023) * 57.2
```

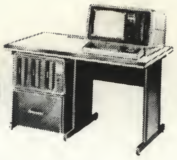
Prints the indicated value.

```
R = N * ATN(-20 * F2/F1)
```

Assigns the indicated value to R.

Sample Program

```
190 INPUT "TANGENT"; T
200 PRINT "ANGLE IS" ATN(T) * 57.29578
```



CDBL
Convert to Double-Precision

CDBL (*number*)
where *number* is any numeric expression.

Returns a double-precision representation of the argument. The value returned will contain 17 digits, but only as many digits as are contained in the argument will be significant.

CDBL may be useful when you want to force an operation to be done in double-precision, even though the operands are single precision or even integers. For example, CDBL (I%)/J% will return a fraction with 17 digits of precision.

Examples

```
Y# = CDBL(N * 3) + M
```

The operations on the right are forced double-precision.

Sample Program

```
210 FOR I = 1 TO 25
220 PRINT 1/CDBL(I),
230 NEXT I
```

Prints the elements of the harmonic series 1, 1/2, 1/3, . . . 1/25 in double-precision.

CINT
Convert to Integer Representation

CINT (*number*)
number is a numeric expression such that −32768 <= *number*<32768.

CINT returns the largest integer not greater than the argument. For example, CINT(1.5) returns 1; CINT(−1.5) returns −2. The result is a two-byte integer.

Examples

```
PRINT CINT(15.0075)
```

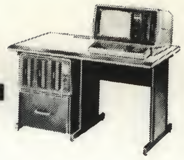
Prints the indicated value.

```
K = CINT(X#) + CINT(Y#)
```

The addition will involve only integer arithmetic, which is much faster than double-precision.

Sample Program

```
240 INPUT "ENTER A POSITIVE DECIMAL NUMBER
(LIKE DDDD.DDDD)"; N
250 PRINT "INTEGER PORTION IS"; CINT(N)
```



COS
Compute Cosine

COS (*number*)
number is a numeric expression.

COS returns the cosine of the angle *number*. The angle must be given in radians. When *number* is in degrees, use COS (*number* * .01745329).

The result is always single-precision.

Examples

```
Y = COS(X)
```
Assigns the value of COS(X) to Y.

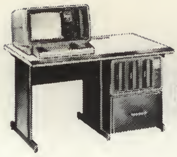
```
Y = COS(X * .01745329)
```
If X is an angle in degrees, the above line will give its cosine.

```
PRINT COS(5.8) - COS(85 * .42)
```

```
G2 = G1 * ((COS(A)) ^ 15)
```

Sample Program

```
260 INPUT "ANGLE IN RADIANS"; A
270 PRINT "COSINE IS" COS(A)
```



CSNG
Convert to Single-Precision

CSNG (*number*)
number is a numeric expression.

CSNG returns a single-precision representation of the argument. When the argument is a double-precision value, it is returned as seven significant digits. When the number is Printed, only six digits will be output, with "4/5" rounding in the least significant digit. For instance, CSNG(.6666666666666667) Prints as .666667; CSNG(.3333333333333333) Prints as .333333.

Examples

```
FC = CSNG(TM#)
```

Assigns the value CSNG (TM#) to FC.

```
PRINT CSNG(.1453885509)
```

Prints a single-precision value.

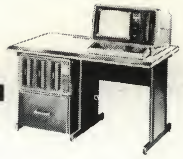
```
R = CSNG(A#/B#)
```

Performs the indicated computation and stores it in R.

Sample Program

```
280 PI# = 3.141592653589793
290 B#  = 18.000000795
300 PRINT CSNG(PI# * B#)
```

This program prints a single-precision value after the double-precision multiplication.



EXP
Compute Natural Exponential

EXP (*number*)
number is a numeric expression.

Returns the natural exponential of *number*, that is, e^{number}. This is the inverse of the LOG function; therefore, X = EXP(LOG(X)). The result is always single-precision.

Examples

```
H = EXP(A)
```

Assigns the value of EXP(A) to H.

```
PRINT EXP(-2)
```

Prints the value .135335.

```
E = (G1 + G2 - .07) * EXP(.055 * (G1 + G2))
```

Performs the required calculation and stores it in E.

Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

FIX
Return Truncated Value

FIX (*number*)
  number is a numeric expression.

FIX returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is a whole number. For non-negative X, FIX(X) = INT(X). For negative values of X, FIX(X) = INT(X) + 1.

The result has the same precision as the argument (except for the fractional portion).

Examples

```
Y = FIX(X)
```

The truncated number is put in Y.

```
PRINT FIX(2.2)
```

Prints the value 2.

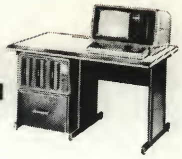
```
PRINT FIX(-2.2)
```

Prints the value -2.

Sample Program

```
330 INPUT "NUMBER"; A#
340 Y# = ABS(A# - FIX(A#))
350 PRINT "FRACTIONAL PORTION IS" Y#
```

This program splits any number into its integer and fractional parts. Try inputting double-precision values.



INSTR
Search for Specified String

INSTR (*position, string 1, string 2*)
position specifies the position in *string 1* where the search is to begin. *position* is optional; if it is not supplied, search automatically begins at the first character in *string 1*. (Position 1 is the first character in *string 1*.)
string 1 is the string to be searched.
string 2 is the substring you want to search for.

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, zero is returned. Note that the entire substring must be contained in the search string, or zero is returned. Also, note that INSTR only finds the first occurrence of a substring starting at the position you specify.

Examples

In these examples, A$ = "LINCOLN":

```
    INSTR(A$, "INC")
```

returns a value of 2.

```
    INSTR (A$, "12")
```

returns a zero.

```
    INSTR(A$, "LINCOLNABRAHAM")
```

returns a zero. For a slightly different use of INSTR, look at

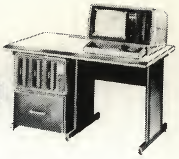
```
   INSTR (3, "1232123", "12")
```

which returns 5.

Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761--) and returns that number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
380 ON ERROR GOTO 410
390 READ ADDRESS$
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER = COUNTER + 1
    ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX ADDRESSES IS" COUNTER:
    END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX *76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD *20760"
440 DATA "1000 TWO TANDY CENTER, FORT  WORTH, TX *76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY, RICHARDSON, TX *75080"
```



INT
Convert to Integer Value

INT(*number*)
number is any numeric expression.

INT returns the largest whole number that is not greater than the argument. The result has the same precision as the argument except for the fractional portion. The argument is **not** limited to the range −32768 to 32767.

Examples

```
A = INT(X)
```

Gets the integer value of X and stores it in A.

```
PRINT INT(2.5)
```

Prints the value 2.

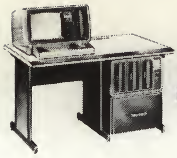
```
PRINT INT(-2.5)
```

Prints −3.

Sample Program

```
460 INPUT X#
470 IF X# < 0 THEN GOTO 460
480 A = INT((X# * 100) + .5)/100
490 PRINT A
```

If you type in a positive number with a fraction like 25.733720, this program will round it off to two decimal places and print it.



LEN
Get Length of String

LEN (*string*)
    string is a string expression.

LEN returns the number of characters in the specified string.

Examples

```
X = LEN(SENTENCE$)
```

Gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

Prints the value 17.

Sample Program

```
500 A$ = ""
510 B$ = "TOM"
520 PRINT A$, B$, B$ + B$
530 PRINT LEN(A$), LEN(B$), LEN(B$ + B$)
```

When this short program is run, the following will be printed on the Display:

```
              TOM            TOMTOM
 0             3              6
```

LOG
Compute Natural Logarithm

LOG (*number*)
  number is a numeric expression.

LOG returns the natural logarithm of the argument. This is the inverse of the EXP function, so X = LOG(EXP(X)). To find the logarithm of a number to another base *B*, use the formula LOG B(X) = LOG E(X)/LOG E(B). For example, LOG(32767)/LOG(2) returns the logarithm to base 2 of 32767.

The result is always single-precision.

Examples

```
B = LOG(A)
```

Computes the value of LOG(A) and stores it in B.

```
PRINT LOG(3.14159)
```

Prints the value 1.14473.

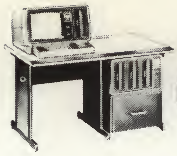
```
Z = 10 * LOG(P2/P1)
```

Performs the indicated calculation and assigns it to Z.

Sample Program

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL (MILES)"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE IS" L "DECIBELS."
```



RND
Generate Pseudorandom Number

RND (*number*)
number is a numeric expression such that
0<=*number*<32768.

RND produces a pseudorandom number using the current "seed" number. The seed is generated internally and is not accessible to the user. RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND (0) returns a single-precision value between 0 and 1. RND(X), where X is an integer between 1 and 32767, returns an integer between 1 and X. For example, RND(55) returns a pseudorandom integer between 1 and 55. RND(55.5) returns a number in the same range, because RND uses the integer value of the argument.

Examples

```
A = RND(2)
```

A is given a value of 1 or 2.

```
A = RND(Z)
```

Returns a random integer between 1 and Z and assigns it to A.

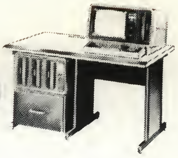
```
PRINT RND(0)
```

Prints a decimal fraction between 0 and 1.

Sample Program

```
580 FOR I = 1 TO 100
590 PRINT RND(10);
600 NEXT I
```

This prints 100 pseudorandom numbers between 1 and 10.



SGN
Get Sign

SGN (*number*)
number is a numeric expression.

This is the "sign" function. It returns −1 if its argument is a negative number, 0 if its argument is zero, and 1 if its argument is a positive number.

Examples

```
Y = SGN(A * B)
```

The function determines what the sign of the expression A * B is, and passes the appropriate number (−1, 0, 1) to Y.

```
PRINT SGN(N)
```

Prints the appropriate number on the Display.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SIN
Compute Sine

SIN (*number*)
number is a numeric expression.

SIN returns the sine of the argument, which must be in radians. To obtain the sine of X when X is in degrees, use SIN(X * .01745329).
The result is always single-precision.

Examples

```
W = SIN(MX)
```

Assigns the value of SIN (MX) to W.

```
PRINT SIN(7.96)
```

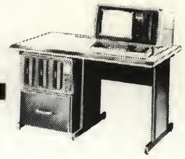
Prints the value .994385.

```
E = (A * A) * (SIN(D)/2)
```

Performs the indicated calculation and stores it in E.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN(A * .01745329)
```



SQR
Compute Square Root

SQR (*number*)
number is a non-negative numeric expression.

SQR returns the square root of its argument. The result is always single-precision.

Examples

```
Y = SQR(A + B)
```

Performs the required calculation and stores it in Y.

```
PRINT SQR(155.7)
```

Prints the value 12.478.

Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

TAN
Compute Tangent

TAN (*number*)
number is a numeric expression.

TAN returns the tangent of the argument. The argument must be in radians. To obtain the tangent of X when X is in degrees, use TAN (X* .01745329). The result is always single-precision.

Examples

```
L = TAN(M)
```

Assigns the value of TAN(M) to L.

```
PRINT TAN(7.96)
```

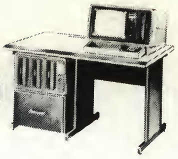
Prints the value −9.39702.

```
Z = (TAN(L2 - L1))/2
```

Performs the indicated calculation and stores the result in Z.

Sample Program

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```



VAL
Evaluate String

VAL (*string*)
string is a string expression.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10^34.

VAL terminates its evaluation on the first character which has no meaing in a numeric term – e.g., Z, ?, etc. The current value at termination is used.

If the string is non-numeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

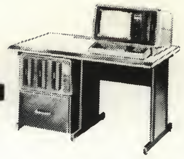
prints 1.234E+08.

```
B = VAL("3" + "*" + "2")
```

The value 3 is assigned to B.

Sample Program

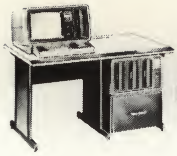
```
750 REM      WHAT SIDE OF THE STREET?
760 REM      NORTH IS EVEN; SOUTH IS ODD
770 LINE INPUT "ENTER THE ADDRESS (NUMBER AND STREET) "; AD$
780 C = INT(VAL(AD$)/2) * 2
790 IF C = VAL(AD$) THEN PRINT "NORTH SIDE": GOTO 770
800 PRINT "SOUTH SIDE": GOTO 770
```



String Functions

All of these functions return a string value. You can use them anywhere a string expression is called for. Notice that several numeric number-related functions (HEX$, OCT$, STR$) are included.

Keyword	Purpose
CHR$	Get specified character
DATE$	Get today's date
HEX$	Convert decimal value to hexadecimal string
LEFT$	Get left portion of string
MID$	Get mid-portion of string
OCT$	Convert decimal value to octal string
RIGHT$	Get right portion of string
SPACE$	Return string of spaces
STR$	Convert to string type
STRING$	Return string of characters
TIME$	Get the time



CHR$
Get Character for ASCII or Control Code

CHR$ (*number*)
number is a numeric expression, in the range [0,255]

CHR$ is the inverse of the ASC function. It returns a one-character string; this character has the ASCII, control, or graphics code number specified by the argument of the function.

Examples:

```
P$ = CHR$(T)
```

The function CHR$ converts the number T into its ASCII character equivalent and puts the character into P$.

```
PRINT CHR$(35)
```

Prints a # on the Display.

```
PRINT CHR$(26)
```

Puts the Display into its black-on-white mode (use CHR$ (25) to return to normal).

```
A$ = A$ + CHR$(I)
```

The character whose ASCII code is I is added to the end of A$.

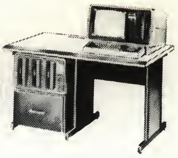
Sample Programs

Using CHR$, you can assign quotation marks to strings, even though they are ordinarily used as string-delimiters. Since the ASCII code for quotations is 34, A$ = CHR$(34) assigns the value " to A$.

```
700 A$ = CHR$(34)
710 PRINT "HE SAID, "; A$; "HELLO."; A$
```

When this is run, the following line will be printed on the Display:

```
HE SAID, "HELLO."
```

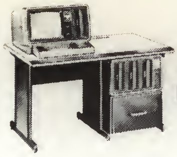


The following program will let you investigate the effect of printing each of the 256 (0-255) codes on the display. Codes of special interest:

1	Turns cursor back on
2	Turns off cursor
4	Starts steady cursor
25	Returns display to normal
26	Reverses display (subsequent printing is black on white)
27	Clears display
30	Starts 80 character/line mode and clears screen
31	Sets 40 character/line mode and clears screen
128-159	Graphics characters

```
100 CLS
110 INPUT"TYPE IN THE CODE (0-255)";C
120 PRINT CHR$(C); "   JUST PRINTED CODE"C
130 GOTO 110
```

For a complete list and discussion of output to the Video Display, see Chapter 1 and the Character Code table in the Appendix.



DATE$
Get Today's Date

DATE$

This function lets you display today's date and use it in a program.

The operator sets the date initially when TRSDOS is started up. When you request the date, BASIC will display it in this fashion:

```
SATAPR281979118 45
```

which means Saturday, April 28, 1979, 118th day of the year, 4th month of the year, 5th day of the week (Monday is the 0th day of the week).

Example

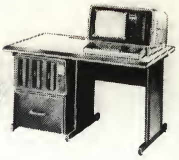
```
PRINT DATE$
```

which returns

```
WEDJUL251979206 72
```

Sample Program

```
1090 PRINT "Inventory Check:"
1100 IF DATE$ = "THUJAN311980031 13" THEN PRINT "Today is
the last day of January 1980.  Time to perform monthly
inventory.": END
1110 A$ = LEFT$(DATE$, 8): B$ =  RIGHT$(A$, 2)
1120 B = VAL(B$)
1130 PRINT 31 - B " days until inventory time."
```



HEX$
Compute Hexadecimal Value

HEX$ (*number*)
  number is a numeric expression in the range [−32768, 32767]

HEX$ returns a string which represents the hexadecimal value of the argument. The value returned is like any other string – it cannot be used in a number expression. That is, you cannot add hex strings. You **can** concatenate them, though.

Examples:

```
    PRINT HEX$(30), HEX$(50), HEX$(90)
```

prints the following strings:

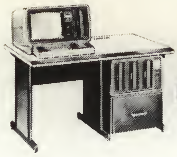
```
    1E      32      5A
```

```
    Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

Sample Program

```
    720 INPUT "DECIMAL VALUE"; DEC
    730 PRINT "HEXADECIMAL VALUE IS " HEX$(DEC)
```



LEFT$
Get Left Portion of String

LEFT$ (*string, number*)
string is a string expression, *string* not equal to null string
number is a numeric expression

LEFT$ returns the first *number* characters of *string*. If *number is equal to or greater than LEN (string)*, the entire string is returned.

Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

Prints the left six characters of BATTLESHIPS, namely, BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

Since BIG FIERCE DOG is less than 20 characters long, the whole phrase is printed.

```
PHRASE$ = LEFT$(M$, 12)
```

Puts the first 12 characters of M$ into PHRASE$.

```
PRINT LEFT$("ALPHA" + "BETA" + "GAMMA", 8)
```

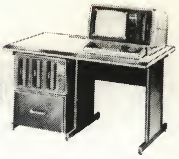
Prints ALPHABET.

Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When this is run, the following will be printed:

```
TIM–THAT'S SHORT FOR TIMOTHY
```



MID$
Get Substring

MID$ (*string, position, length*)
  string is a string expression
  position is the position where the substring begins in string
  length is the number of characters in the substring (this parameter is optional). If omitted, LEN (*string*) is used.

MID$ returns a substring of *string*. The substring begins at *position* in *string* and is *length* characters long.

Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

OCT$
Compute Octal Value

OCT$(*number*)
number is a numeric expression in the range [−32768, 32767]

OCT$ returns a string which represents the octal value of the argument. The value returned is like any other string – it cannot be used in a numeric expression.

Examples:

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the following strings:

```
36       62       132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

Sample Program

```
830 INPUT "DECIMAL VALUE"; DEC
840 PRINT "OCTAL VALUE IS " OCT$(DEC)
```

RIGHT$
Get Right Portion of String

RIGHT$ (*string, number*)
string is a string expression, *string* not equal to null string
number is a numeric expression

RIGHT$ returns the last *number* characters of *string*. If LEN (*string*) is less than or equal to *number*, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

Prints the five right characters of WATERMELON, namely, MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

Since MILKY WAY is less than 25 characters long, the whole phrase is printed.

```
ZIP$ = RIGHT$(ADDRESS$, 5)
```

Puts the last five characters of ADDRESS$ into ZIP$.

```
PRINT RIGHT$(STRING$(50, CHR$(33)), 1)
```

Prints a single "!".

Sample Program

```
850 RESTORE: ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2);: GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMOND MANUFACTURING COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

SPACE$
Return String of Spaces

SPACE$ (*length*)
length is a numeric expression, in the range [0,255].

SPACE$ returns a string of spaces. The number of spaces is determined by the argument.

Examples:

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9) "QUANTITY"
```

Prints DESCRIPTION followed by four spaces followed by TYPE followed by nine spaces followed by QUANTITY.

```
A$ = SPACE$(14)
```

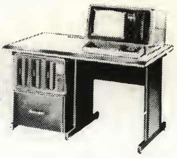
Puts a string of fourteen spaces into A$.

```
SP$ = SPACE$(N)
```

Puts a string of N spaces into SP$.

Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```



STR$
Convert to String Representation

STR$(*number*)
number is a numeric expression.

STR$ converts its argument to a string. For example, if X = 58.5, then STR$ (X) equals the string “ƀ58.5. Notice that a leading blank is inserted before 58.5 to allow for the sign of X. While arithmetic operations may be performed on X, only string function and operations may be performed on the string, “ƀ58.5”.

Examples:

```
S$ = STR$(X)
```

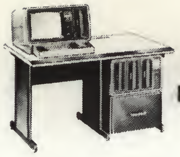
Converts the number X into a string and stores it in S$.

```
T$ = STR$(A * 18)
```

Converts the number A * 18 into a string and stores it in T$.

Sample Program

```
90 PRINT "THIS PROGRAM DEMONSTRATES HOW TO PREVENT AN ERROR"
95 PRINT "FROM ENTERING INTO A SINGLE-TO-DOUBLE PRECISION CONVERSION"
97 PRINT
100 A=1.6
110 B#=A
120 C#=VAL(STR$(A))
130 PRINT"REGULAR CONVERSION"TAB(40)"SPECIAL CONVERSION"
140 PRINT B# TAB(40) C#
```



STRING$
Return String of Characters

STRING$ (*length, character*)
  length is a numeric expression in the range [0,255].
  character is a string expression or an ASCII code.
  If a string constant is used, it must be enclosed in quotes.

STRING$ returns a string of characters. How many characters are returned depends on STRING$'s first argument; what characters they are depends on its second argument. For example, STRING$(30,65) returns a string of 30 "A"s. STRING$(30,20) returns a string of 30 blanks, since 20 is the code for a blank character.

STRING$ is useful for creating graphs, tables, and so on.

Examples:

```
B$ = STRING$(25, "X")
```

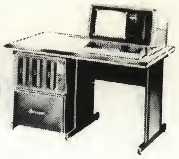
Puts a string of 25 "X"s into B$.

```
PRINT STRING$(50, 10)
```

10 is ASCII code for a line feed, so the line above will print 50 blank lines on the Display.

Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN THREE NUMBERS BETWEEN 33 AND 159 (N1, N2
, N3)"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20, N1): NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2): NEXT J
1080 PRINT STRING$(80, N3)
```



TIME$
Get the Time

```
TIME$
```

This function lets you use the time in a program.

The operator sets the time initially when TRSDOS is started up. When you request the time, TIME$ will supply it using this format:

```
14.47.18
```

which means 14 hours, 47 minutes, and 18 seconds (24-hour clock) or 2:47:18 PM

To change the time, use the TRSDOS command, TIME. For example,

```
SYSTEM "TIME 13.30"
```

Example

```
A$ = TIME$
```

When this line is reached in your program, the current time is stored in A$.

Sample Program

```
1140 IF LEFT$(TIME$, 5) = "10.15" THEN PRINT "Time is 10:15
A.M.--time to pick up the mail.": END
1150 GOTO 1140
```

Input/Output Functions

These functions perform input/output to the keyboard, video display, line printer, and disk files. They are grouped accordingly in this section.

Functions with a $ suffix (like INKEY$) return string values; others return numeric values.

Note: Before attempting any input/output to BASIC data files, you should read Chapter 4, **File Access Techniques**, and try out the sample programs given there.

Keyword	Purpose
INKEY$	Get keyboard character if available
INPUT$	Get a string of characters from keyboard
POS	Get cursor column position on video display
ROW	Get cursor row position on video display
SPC	Output spaces to video display
CVD	Restore data from disk file to double-precision (direct access)
CVI	Restore data from disk file to integer (direct access)
CVS	Restore data from disk file to single-precision (direct access)
EOF	Check for end of file
INPUT$	Input a string of characters from disk file (sequential access)
LOC	Get current disk file record number (direct or sequential access)
LOF	Get disk file's end of file record number
MKD$	Make double-precision number to string for disk write (direct access)
MKI$	Make integer to string for disk write (direct access)
MKS$	Make single-precision number to string for disk write (direct access)

INKEY$
Get Keyboard Character if Available

```
INKEY$
```

Returns a one-character string from the keyboard without the necessity of having to press **ENTER**. If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the Display.

INKEY$ is invariably put inside some sort of loop. Otherwise program execution would pass through the line containing INKEY$ before a key could be pressed.

Example

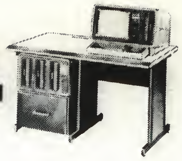
```
A$ = INKEY$
```

When put into a loop, the above program fragment will get a key from the keyboard and store it in A$. If the line above is used by itself, when control reaches it and no key is being pressed, a null string (" ") will be stored in A$.

Sample Program

```
1200 CLS
1210 PRINT@ 540, INKEY$;
1220 GOTO 1210
```

When you run this program, the screen will remain blank (except for the cursor) until you strike a key. The last key that you strike will remain on the Display until you press another one. Whenever you fail to hit a key while this program is executing, a null string, i.e., nothing, is printed at 540.



INPUT$
Input a Character String

INPUT$ (*length*)
  length is a numeric expression in the range [1,255].

This function allows a program to input a specified number of keyboard characters. As soon as the last required character is typed, execution continues. (You don't have to press **ENTER** to signify end-of-line.) The characters you type will not be displayed on the screen.

Any character you type will be accepted (except **BREAK**).

Examples

```
A$=INPUT$(5)
```

A string of 5 characters must be input before BASIC will proceed to the next line of the program.

Sample Program

This program shows how you might use INPUT$ to have an operator input a password to access a protected file. By using INPUT$, the operator can type in the password without anyone see it on the Video Display. (To see the full file specification, run the program, then type PRINT F$.)

```
110 LINE INPUT "TYPE IN THE FILENAME/EXT"; F$
120 PRINT "TYPE IN THE PASSWORD -- MUST TYPE 8 CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ + "." + P$
```

POS
Get Cursor Column Position

POS (*dummy*)
  dummy is any numeric expression.

POS returns a number from 0 to 79 indicating the current cursor column-position on the Display.

Examples

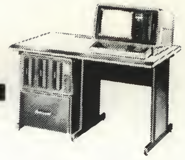
```
PRINT TAB(40) POS(0)
```

The PRINT TAB statement moves the cursor to position 40. Since the cursor is at 40, POS(0) returns the value 40, and 40 is printed on the Display. (However, since a blank is inserted before the "4" to accommodate the sign, the "4" is actually at position 41.) The "0" in POS(0) is the dummy argument.

Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) > 70 THEN IF A$ = CHR$(32) THEN A$ = CHR$(13)
190 PRINT A$;
200 LPRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that mistakes can't be corrected). Your computer keyboard is the typewriter keyboard. The program will keep watch at the end of a line so that no word is divided beween two lines.



ROW
Get Row Position of Cursor

ROW (*dummy*)
dummy is a dummy argument.

The ROW function finds the row on which the cursor is currently located and returns that row-number. The 24 rows are numbered 0-23.

Examples

```
X = ROW(Y)
```

The row-number of the cursor's position at the time this line is encountered is assigned to X.

```
PRINT ROW(0)
```

The row-number is printed on the Display.

Sample Program

When a key is typed, the program below will print it, find its Display row-number and column-number, print this information, find its ASCII code, and print this information too.

```
100 CLS
110 R=0:C=0
120 PRINT@(21,32),"ROW","COLUMN"
130 X$=INPUT$(1)
140 PRINT @(R,C),X$;
150 C=POS(0):R=ROW(0)
160 PRINT @ (22,32),R,C;
163 PRINT @ (23,32),STRING$(20,32);
165 PRINT @(23,32),"ASCII CODE IS "HEX$(ASC(X$));
170 PRINT@(R,C),"";
180 GOTO 130
```

SPC
Print Line of Blanks

SPC (*number*)
number is a numeric expression,
number = 0,1,2, . . ., 255

SPC prints a line of blanks. The number of blanks is determined by the argument of SPC. SPC does not use string space.

The left parenthesis must immediately follow SPC (no blanks in between).

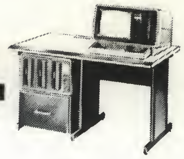
SPC can be used with PRINT, LPRINT, or PRINT# statements.

Examples

```
PRINT SPC(25); "Hello"
```

Sample Program

```
1250 PRINT SPC(75) "Here"
1260 PRINT SPC(60) "is"
1270 PRINT SPC(45) "an"
1280 PRINT SPC(30) "example"
1290 PRINT SPC(15) "of"
1300 PRINT "SPC"
```



CVD, CVI, CVS
Restore String Data to Numeric

CVD (*string*)
string is a string expression which defines an eight-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN(*string*) <8, an Illegal Function Call occurs; if LEN(*string*)>8, only the first eight characters are used.

CVI (*string*)
string is a string expression which defines a two-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN*string*) <2, an Illegal Function Call occurs; if LEN*string*)>2, only the first two characters are used.

CVS (*string*)
string is a string expression which defines a four-character string; *string* is typically the name of a buffer-field containing a numeric string. If LEN(*string*) <4, an Illegal Function Call occurs; if LEN(*string*)>4, only the first four characters are used.

These functions let you restore data to numeric form after it is read from disk. Typically the data has been read by a GET statement, and is stored in a direct access file buffer. CVD, CVI, and CVS are the inverses of MKD$, MKI$, and MKS$, respectively.

Examples

Suppose the name GROSSPAY$ references an eight-byte field in a direct access file buffer, and after GETting a record, GROSSPAY$ contains an MKD$ representation of the number 13123.38. Then the statement

```
A# = CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

Sample program

```
1420 OPEN "D", 1, "TEST/DAT"
1430 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1440 GET 1
1450 PRINT CVI(I1$), CVS(I2$), CVD(I3$)
1460 CLOSE
```

This program opens a file named TEST/DAT which is assumed to have been previously created. (For the program which creates the file, see the section on MKD$, MKI$, and MKS$.) CVI, CVS, and CVD are used to convert string data back to numeric form.

EOF
End-of-file detector

EOF (*buffer-number*)

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid Input Past End errors during sequential input.

Assuming *buffer-number* specifies an open file-buffer, then EOF (*buffer-number*) returns 0 (false) when the EOF record has not yet been read, and −1 (true) when it has been read.

Examples

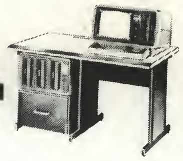
```
IF EOF(FILE) THEN CLOSE FILE
```

This line determines whether the end-of-file has been reached. If it has, the specified buffer (FILE) is closed.

Sample program

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 “passes”, so the program branches out of the disk access loop, preventing an Input Past End error from occurring. Also note that the variable I contains the number of elements input into array A( ).

```
1470 DIM A(100)       'ASSUMING THIS IS A SAFE VALUE
1480 OPEN "I", 1, "DATA/TXT"
1490 I% = 0
1500 IF EOF(1) THEN GOTO 1540
1510 INPUT#1, A(I%)
1520 I% = I% + 1
1530 GOTO 1500
1540 REM   PROG. CONT. HERE AFTER DISK INPUT
```



INPUT$
Input Specified Number of Bytes from Disk

INPUT$ (*length, buffer-number*)
  length is the number of bytes to be input
  buffer-number is a sequential input file buffer (1,2,3, . . .15)

This function is analogous to keyboard INPUT$ except that it inputs data from disk rather than the keyboard.

You can use disk INPUT$ to get a certain specified number of bytes (sequential access only). INPUT$, in contrast to INPUT#, allows you to get any number of data bytes (up to 255) from disk.

Example

```
A$ = INPUT$(12, 2)
```

Inputs 12 bytes from disk into A$. File-buffer 2 is used.

Sample Program

```
2200 OPEN "I", 1, "TEST/DAT"
2210 T$ = INPUT$(70, 1)
2220 CLOSE
```

If a file TEST/DAT has been created previously, this program will open it, retrieve 70 bytes from it, store the data in T$, and close the file.

LOC
Get Current Record Number

LOC (*file number*)
  file number is a numeric expression specifying the buffer for a currently-open file

LOC is used to determine the current record number, i.e., the number of the last record processed since the file was opened. It returns the record number that will be used if a GET or PUT is executed with the record number omitted.

LOC is also valid for sequential files, and gives the number of 1-byte records processed since the OPEN statement was executed.

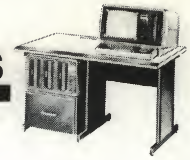
Example

```
PRINT LOC(1)
```

Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD" LOC(1): CLOSE: END
1340 GOTO 1320
```

This is a portion of a program. Elsewhere the file has been opened and fielded. N$ is a field variable. If N$ matches A$ the record number in which it was found is printed.



LOF
Get End-of-File Record Number

LOF (*number*)
  number specifies a random access buffer,
    number = 1,2, . . .,15

This function tells you the number of the last, i.e., highest-numbered, record in a file. It is useful for both sequential and direct access.

Examples

```
Y = LOF(5)
```

Puts the record number into variable Y.

Sample Programs

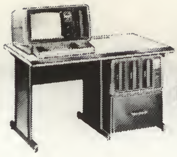
During direct access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

```
1540 OPEN "R", 1, "UNKNOWN/TXT"
1550 FIELD 1, 255 AS A$
1560 FOR I% = 1 TO LOF(1)  'LOF(1) = HIGHEST REC-
1570 GET 1,I%              'ORD NUM. TO BE ACCESSED
1580 PRINT A$
1590 NEXT I%
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an End of File error.

When you want to add to the end of a file, LOF tells you where to start adding:

```
1600 I% = LOF(1) + 1     'HIGHEST EXISTING RECORD
1610 PUT 1,I%            'ADD NEXT RECORD
```



MKD$, MKI$, MKS$
Convert Numeric to String

MKD$(*number*)
MKI$(*number*)
MKS$(*number*)
number is a numeric expression.

These three functions are the inverses of CVD, CVI, and CVS. They change a number to a string. Actually, the byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

MKD$ returns an eight-byte string; MKI$ returns a two-byte string; and MKS$ returns a four-byte string.

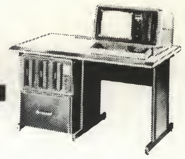
Examples

```
    LSET AVG$ = MKS$(0.123)
```

Sample Program

```
    1350 OPEN "D", 1, "TEST/DAT"
    1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
    1370 LSET I1$ = MKI$(3000)
    1380 LSET I2$ = MKS$(3000.1)
    1390 LSET I3$ = MKD$(3000.00001)
    1400 PUT 1
    1410 CLOSE
```

For a program the retrieves the data from TEST/DAT , see CVD/CVI/CVS.



Special Functions

With the special functions you can perform memory-related tasks like finding or changing the amount of total memory or string space, and discovering the absolute memory address of the value of a variable.

For example:

```
S$ = FRE(A$)
```

will find the number of bytes of string storage space left, and put this value in S$.

Other special functions, such as VARPTR and USR*n*, let you interface your BASIC program with machine-language programs.

Keyword	Purpose
FRE	Get amount of free memory or string space
MEM	Get amount of free memory
USRn	Call machine-language subroutine
VARPTR	Get absolute memory address

FRE
Get Amount of Memory/String Space

FRE (*dummy*)
dummy is a numeric dummy argument, or a string dummy argument.

FRE returns two different values depending on its argument. If the argument is a number or numeric variable, FRE will return the total amount of memory available. If the argument is a string or string variable, FRE will return the total amount of string storage space that is available.

Examples

```
PRINT FRE(44)
```

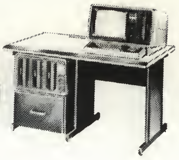
Prints the amount of memory left.

```
PRINT FRE("44")
```

Prints the amount of string space left.

Sample Program

```
10 PRINT"CURRENT FREE STRING SPACE IS"FRE(A$)
20 LINE INPUT"TYPE IN A MESSAGE-- ";M$
30 PRINT"AFTER STORING MESSAGE, FREE STRING SPACE IS" FRE(A$)
```



MEM
Get Amount of Memory

```
MEM
```

MEM performs the same function as FRE when FRE is followed by a numeric dummy argument. MEM returns the number of unused and unprotected bytes in memory. This function may be used in the immediate mode to see how much space a resident program occupies, or it may be used inside a program to avert out of memory errors by allocating less string space and dimensioning smaller array sizes. MEM requires no argument.

Example

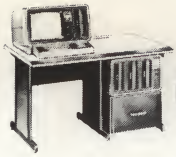
```
PRINT MEM
```

Enter this command (in the immediate mode; no line number is needed). The number returned indicates the amount of leftover memory, i.e., memory not being used to store programs, variables, strings, the stack, or not reserved for object files.

Sample Program

```
1610 IF MEM < 80 THEN 1630
1620 DIM A(15)
1630 REM     PROGRAM CONTINUES HERE
```

If fewer than 80 bytes of memory are left, control switches to another part of the program. Otherwise, an array of 15 elements is created.



USRn
Call User's External Subroutine

USR*n* (*argument*)
n specifies one of ten available USR calls, *n* = 0,1,2, . . .,9.
If *n* is omitted, zero is assumed.
argument is a numeric or string expression.

These functions (USR0 through USR9) let you call as many as 10 machine-language subroutines and then continue execution of your BASIC program. These subroutines must have been previously defined with DEFUSR*n* statements.

"Machine language" is the low-level language used internally by your Computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and for doing things very fast (like white-out the Display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. For more information on this subject, see the Radio Shack book, *TRS-80 Assembly-Language Programming,* by William Barden, Jr.

When a USR call is encountered in a statement, control goes to the address defined in the DEFUSR*n* statement. This address specifies the entry point to your machine-language routine.

Examples

```
X = USR5(Y)
```

When this statement is executed, BASIC calls the machine-language routine USR5, previously defined in a DEFUSR5 = *address* statement.

Passing arguments from BASIC to the subroutine:
Upon entry to a USR*n* subroutine, the following register contents are set up (for notation, see page 86 of the **TRSDOS Reference Manual).**

A = Type of argument in USR*n* reference
A = 8 if argument is double-precision.
A = 4 if argument is single-precision.
A = 2 if argument is integer.
A = 3 if argument is string.

HL = When the argument is a number, this register points to the argument storage area (ASA) described later.

DE = When the argument is a string, this register points to a string descriptor, as follows:
The first byte gives the length of the string. The next two bytes give the address where the string is stored: least significant byte (LSB) followed by most significant byte (MSB).

Description of Argument Storage Area (ASA)—for numeric values only.

For double-precision numbers:

ASA	Exponent in 128-excess form. E.g., a value of 200 indicates a 0 exponent; a value of 128 indicates a −62 exponent. A value of 0 always indicates the number is zero.
ASA-1	Highest 7 bits of the mantissa with hidden (implied) leading one. Bit 7 is the sign of the number (0 positive, 1 negative). E.g., a value of X'84' indicates the number is negative and the MSB of the mantissa is X'84'. A value of X'04' indicates the number is positive and the MSB of the mantissa is X'84'.
ASA-2 through ASA-6	Successive 8-bit blocks of the mantissa.
ASA-7	Lowest 8 bits of the mantissa.

For single-precision numbers:

ASA through ASA-3 same as for double-precision numbers.

For integer numbers:

ASA	LSB of the number.
ASA+1	MSB of the number. Together, the two bytes represent the number in signed, two's complement form.

To convert the argument to integer type:

Your routine can call BASIC's FRCINT routine to put the argument into HL in 16-bit, signed two's complement form. The address of FRCINT is stored in [X'2803', X'2804'].

For example, you can put the following code at the beginning of your subroutine:

```
FRCINT  EQU     2803H           ;CONVERTS USR ARGUMENT
                                ; TO INTEGER IN HL
        LD      HL,CTNU         ;(HL)=CONTINUATION
                                ; ADDRESS
        PUSH    HL              ;SAVE IT FOR RETURN
                                ; FROM FRCINT
        LD      HL,(FRCINT)     ;(HL)=FORCE INTEGER
                                ; ROUTINE
        JP      (HL)            ;DO FRCINT ROUTINE
```

Returning values from the subroutine to BASIC

When the USR*n* argument is a variable, you can modify its value by changing the ASA or string contents, as pointed to by HL or DE. For example, the statement:

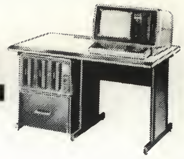
```
    X=USR1(A%)
```

transfers control to the USR1 subroutine, with HL pointing to the two-byte ASA for integer variable A%. Suppose you modify the contents of this storage area. When you do a RET instruction to return to BASIC, A% will have a new value, and X will be assigned this new value.

In general, USR*n*(*argument*) will return the same type of value as *argument*. However, you can use BASIC's MAKINT routine to return an integer value. The address of the MAKINT routine is stored at [X'2805',X'2806'].

For example, you might include the following code at the end of your program to return a value to BASIC.

```
MAKINT  EQU     2805H
        LD      HL,VAL          ;VAL IS THE VALUE TO
                                ;BE RETURNED.
        PUSH    HL              ;SAVE VALUE IN STACK
        LD      HL,(MAKINT)     ;RESTORE VAL INTO HL
        EX      (SP),HL         ;AND PUT MAKINT
                                ;INTO STACK
        RET
```



VARPTR
Gets Absolute Memory Address

VARPTR (*variable name* or *file number*)

VARPTR returns an absolute memory address which will help you locate a value in memory. When used with a variable name, it locates the contents of that variable. When used with a file number, it returns the address of the file's data buffer. If the variable you specify has not been assigned a name, or the file has not been opened, an Illegal Function Call will occur.

VARPTR is used primarily to pass a value to a machine language subroutine via USRn. Since VARPTR returns an address which indicates where the value of a variable is stored, this address can be passed to a machine language subroutine as the argument of USR; the subroutine can then extract the contents of the variable with the help of the address that was supplied to it.

If VARPTR(*integer variable*) returns address K:
Address K contains the least significant byte (LSB) of 2-byte *integer*.
Address K+1 contains the most significant byte (MSB) of *integer*.

If VARPTR(*single precision variable*) returns address K:
(K)* = LSB of value
(K+1) = Next most sig. byte (Next MSB)
(K+2) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number
(K+3) = exponent of value excess 128 (128 is added to the exponent).

If VARPTR(*double precision variable*) returns K:
(K) = LSB of value
(K+1) = Next MSB
(K+ . . .) = Next MSB
(K+6) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number.
(K+7) = exponent of value excess 128 (128 is added to the exponent).

For single and double precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

* (K) signifies "contents of address K"

If VARPTR(*string variable*) returns K:
(K) = length of string
(K+1) = LSB of string value starting address
(K+2) = MSB of string value starting address

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$="HELLO" do not use string storage space.

For all of the above variables, addresses (K-1) and (K-2) will store the TRS-80 Character Code for the variable name. Address (K-3) will contain a descriptor code that tells the Computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

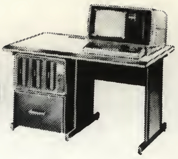
VARPTR(*array variable*) will return the address for the first byte of that element in the array. The element will consist of 2 bytes if it is an integer. array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:

1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.
5. A one-byte type-descriptor (02 = Integer, 03 = String, 04 = Single-Precision, 08 = Double-Precision).

Item 1 immediately precedes the first element, Item 2 precedes Item 1, and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.



Examples

A! = 2 will be stored as follows:
2 = 10 Binary, normalized as .1E2 = .1 x 10^2
So exponent of A is 128+2 = 130 (called excess 128)
MSB of A is 10000000; however, the high bit is changed to zero since the value is positive (called hidden or implied leading one).

So A! is stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
130	0	0	0

A!=−.5 will be stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
128	128	0	0

A!=7 will be stored as

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
131	96	0	0

A!=−7:

Exponent (K+3)	MSB (K+2)	Next MSB (K+1)	LSB (K)
131	224	0	0

Zero is simply stored as a zero-exponent. The other bytes are insignificant.

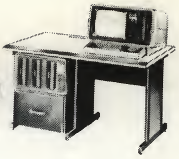
```
Y = USR1(VARPTR(X))
```

If X is an integer value, VARPTR(X) finds the address of the least significant byte of X. This address is passed to the subroutine, which in turn passes its result to Y.

Chapter 4

File Access Techniques

This chapter briefly shows how to "put together" Model II BASIC's many disk-related statements and functions. For syntax and other details on any particular statement or function, see Chapter 3.



Methods of Access

Model II BASIC provides two means of file access:

- Sequential—in which you start reading or writing data at the beginning of a file; subsequent reads or writes are done at following positions in the file.
- Direct—in which you start reading or writing at any record you specify. (Direct access is also called random access, but "direct" is more descriptive.)

Sequential access is stream-oriented; that is, the number of characters read or written can vary, and is usually determined by delimiters in the data. **Direct access is record-oriented;** that is, data is always read or written in fixed-length blocks called records.

Note: When you start BASIC from TRSDOS, you select the maximum number of files you will want to have Open simultaneously. For example, the TRSDOS command line:

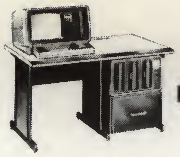
```
TRSDOS READY
BASIC -F:3
```

starts BASIC with a maximum of three concurrent data files, i.e., data files Open simultaneously.

To do any input/output to a disk file, you must first Open the file. When you Open the file, you specify what kind of access you want:

- "O" for sequential output
- "I" for sequential input
- "D" for direct input/output ("R" can also be used)

You also assign a file buffer for BASIC to use during file accesses. This number can be from 1 to 15, but must not exceed the number of concurrent files you requested when you started BASIC from TRSDOS. For example, if you started BASIC with 3 files, you can use buffer numbers 1, 2, and 3. Once you assign a buffer number to a file, you cannot assign that number to another file until you Close the first file.



Examples:

```
OPEN "O", 1, "TEST"
```

Creates a sequential output file named TEST on the first available drive; if TEST already exists, its previous contents are lost. Buffer 1 will be used for this file.

```
OPEN "I", 2, "TEST"
```

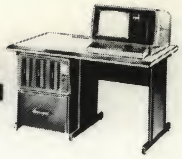
Opens TEST for sequential input, using buffer 2.

```
OPEN "D", 1, "TEST"
```

Opens TEST for direct access, using buffer 1, If TEST does not exist, it will be created on the first available drive. Since record length is not specified, 256-byte records will be used.

```
OPEN "D", 1, "TEST", 40
```

Same as preceding example, but 40-byte records will be used.



Sequential Access

This is the simplest way to store data in and retrieve it from a file. it is ideal for storing free-form data without wasting space between data items. You read the items back in the same order in which they were written.

There are several important points to keep in mind.

1. You must start writing at the beginning of the file. If the data you are seeking is somewhere inside, you have to read your way up to it.

2. Each time you Open a file for sequential output, the file's previous contents are lost.

3. To update (change) a sequential file, read in the file and write out the updated data to a **new** output file.

4. Data written sequentially usually includes delimiters (markers) to signify where each data item begins and ends. To read a file sequentialy, you must know ahead of time the format of the data. For example: Does the file consist of lines of text terminated with carriage returns? Does it consist of numbers separated by blank spaces? Does it consist of alternating text and numeric information?

5. Sequential files are always written as ASCII-coded text, one byte for each character of data. For example, the number:
    b̸1.2345b̸
requires 8 bytes of disk storage, including the leading and trailing blanks that are supplied. The text string:
    Johnson, b̸Robert
requires 15 bytes of disk storage.

6. Sequential files are always written with a record length of one. This matters if you want to Close the file and re-Open it for Direct access; in such a case, you must specify a record length of 1.

Sequential Output: An Example

Suppose we want to store a table of English-to-metric conversion constants:

English unit	Metric equivalent
1 inch	2.54001 centimeters
1 mile	1.60935 kilometers
1 acre	4046.86 sq. meters
1 cubic inch	0.01638716 liter
1 U.S. gallon	3.785 liters
1 liquid quart	0.9463 liter
1 lb (avoir)	0.45359 kilogram

First we decide what the data image is going to be. Let's say we want it to look like this:

english unit–>metric unit, factor X'0D'

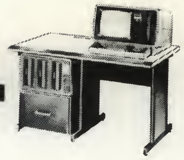
For example, the stored data would start out:

IN–>CM,ƀ2.54001ƀ X'0D'

The following program will create such a data file.

Note: X'0D' represents a carriage return.

```
10 OPEN "O",1,"METRIC/TXT"
20 FOR I%=1 TO 7
30        READ UNIT$, FACTR
40        PRINT #1, UNIT$; ","; FACTR
50 NEXT
60 CLOSE
70 DATA IN->CM, 2.54001, MI->KM, 1.60935, ACRE->SQ.M, 4046.86
80 DATA CU.IN->LTR, 1.638716E-2, GAL->LTR, 3.785
90 DATA LIQ.QT->LTR, 0.9463, LB->KG, 0.45359
```



Line 10 creates a disk file named METRIC/TXT, and assigns buffer 1 for sequential output to that file. The extension /TXT is used because sequential output always stores the data as ASCII-coded text.

Note: If METRIC/TXT already exists, line 10 will cause all its data to be lost. Here's why: Whenever a file is opened for sequential output, the end-of-file (EOF) is set to the beginning of the file. In effect, TRSDOS "forgets" that anything has ever been written beyond this point.

Line 40 prints the current contents of UNIT$ and FACTR to the file. Since the string items do not contain delimiters, it is not necessary to print explicit quotes around them. The explicit comma is sufficient.

Line 60 closes the file. The EOF is at the end of the last data item, i.e., 0.45359, so that later, during input, BASIC will know when it has read all the data.

Sequential Input: An Example

The following program reads the data from METRIC/TXT into two "parallel" arrays, then asks you to enter a conversion problem.

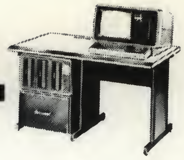
```
5 CLEAR 500
10 DIM UNIT$(9), FACTR(9)        'allows for up to 10 data pairs
20 OPEN"I",1,"METRIC/TXT:1"
25 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1, UNIT$(I%),FACTR(I%)
50 I%=I%+1
60 GOTO 30
70 CLOSE          '      Conversion factors have been read-in
100 CLS: PRINT TAB(5)"*** English to Metric Conversions ***"
110 FOR ITEM%=0 TO I%-1
120     PRINT TAB(9);USING"(## )    \     \";ITEM%, UNIT$(ITEM%)
130 NEXT
140 PRINT @ (19,0), "Which conversion (0-6)";
150 INPUT CHOICE%
160 INPUT"Enter English quantity";V
170 PRINT"The Metric equivalent is" V*FACTR(CHOICE%)
180 INPUT"Press <ENTER> to continue";X
190 PRINT @ (19,0), CHR$(24)     'clear to end of frame
200 GOTO 140
```

Line 20 opens the file for sequential input. Input begins at the beginning of the file.

Line 30 checks to see that the end-of-file record hasn't been reached. If it has, control branches from the disk input loop to the part of the program that uses the newly acquired data.

Line 40 reads a value into the string array UNIT$( ), and a number into the single-precision array FACTR( ). Note that this INPUT list parallels the PRINT# list that created the data file (see the section "Sequential Output: An Example"). This parallelism is not required, however. We could just as successfully have used:

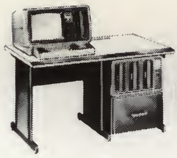
```
40 INPUT#1, UNIT$(I%): INPUT#1,FACTR(I%)
```



How to update a file

Suppose you want to add more entires into the English-Metric conversion file. You can't simply re-Open the file for sequential output and PRINT# the extra data – that would immediately set the EOF to the beginning of the file, effectively destroying the file's previous contents. Do this instead:

1) Open the file for sequential input
2) Open another new data file for sequential output
3) Input a block of data and update the data as necessary
4) Output the data to the new file
5) Repeat steps 3 and 4 until all data has been read, updated, and output to the new file; then go to step 6
6) Close both files



Sequential Line Input: An Example

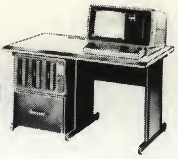
Using the line-oriented input, you can write programs that edit other BASIC program files: renumber them, change LPRINTs to PRINTs, etc.—as long as these "target" programs are stored in ASCII format.

The following program counts the number of lines in any ASCII—format BASIC disk file with the extension /TXT.

```
10 CLEAR 300
20 INPUT"WHAT IS THE NAME OF THE PROGRAM"; PROG$
30 IF INSTR(PROG$,"/TXT")=0 THEN 110 'require /TXT extension
40 OPEN"I", 1, PROG$
50 I%=0
60 IF EOF(1) THEN 90
70 I%=I%+1: LINE INPUT#1, TEMP$
80 GOTO 60
90 PRINT PROG$" IS" I% "LINES LONG."
100 CLOSE: GOTO 20
110 PRINT "FILESPEC MUST INCLUDE THE EXTENSION '/TXT'"
120 GOTO 20
```

For BASIC programs stored in ASCII, each program line ends with a carriage return character not preceded by a line feed. So the LINE INPUT in line 70 automatically reads one entire line at a time, into the variable TEMP$. Variable I% actually does the counting.

To try out the program, first save any BASIC program using the A (ASCII) option (See SAVE). Use the extension /TXT.



Direct Access Techniques

Direct access offers several advantages over sequential access:

- Instead of having to start reading at the beginning of a file, you can read any record you specify.
- To update a file, you don't have to read in the entire file, update the data, and write it out again. You can rewrite or add to any record you choose, without having to go through any of the other records.
- Direct access is more efficient—data takes up less space and is read and written faster.
- Opening a file for direct access allows you to write and read from the file via the same buffer.
- Direct access provides many powerful statements and functions to structure your data. Once you have set up the structure, direct input/output becomes quite simple.

The last advantage listed above is also the "hard part" of direct access. It takes a little extra thought.

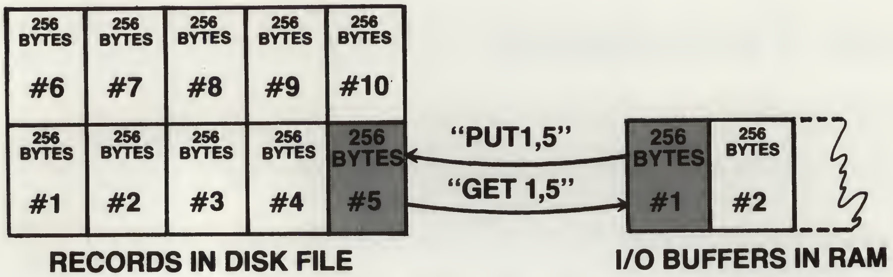
For the purposes of direct access, you can think of a disk file as a set of boxes—like a wall of post-office boxes. Just like the post office receptacles, the file boxes are numbered. We call these boxes "records."

Each record may contain between 1 and 256 bytes. The length of the records is set when you create a file, in the OPEN statement.

You can place data in any record, or read the contents of any record, with statements like:

`PUT 1,5` write buffer-1 contents to record 5
`GET 1,5` read the contents of record 5 into buffer-1

In the following illustration, we assume a record length of 256.



The buffer is a waiting area for the data. Before writing data to a file, you must place it in the buffer assigned to the file. After reading data from a file, you must retrieve it from the buffer.

As you can see from the sample PUT and GET statements above, data is passed to and from the disk in records. The size of each record is determined by an Open statement.

Storing Data in a Buffer

You must place the entire record into the buffer before putting its contents into the disk file.

This is accomplished by 1) dividing the buffer up into fields and naming them, then 2) placing the string or numeric data into the fields.

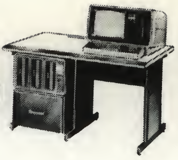
For example, suppose we want to store a glossary on disk. Each record will consist of a word followed by its definition. We start with:

```
100 OPEN"D", 1, "GLOSSARY/BAS"
110 FIELD 1, 16 AS WD$, 240 AS MEANING$
```

Line 100 opens a file named GLOSSARY/BAS (creates it if it doesn't already exist); and gives buffer 1 direct access to the file.

Line 110 defines two fields onto buffer 1:
WD$ consists of the first 16 bytes of the buffer;
MEANING$ consists of the last 240 bytes.

WD$ and MEANING$ are now **field-names.**



What makes field names different? Most string variables point to an area in memory called the string space. This is where the value of the string is stored.

Field names, on the other hand, point to the buffer area assigned in the FIELD statement. So, for example, the statement:

```
  10 PRINT WD$; ":"; MEANING$
```

displays the contents of the two buffer fields defined above.

These values are meaningless unless we first place data in the buffer. LSET, RSET and GET can all be used to accomplish this function. We'll start with LSET and RSET, which are used in preparation for disk output.

Our first entry is the word "left-justify" followed by its definition.

```
  100 OPEN"D", 1, "GLOSSARY/BAS"
  110 FIELD 1, 16 AS WD$, 240 AS MEANING$
  120 LSET WD$="LEFT-JUSTIFY"
  130 LSET MEANING$="To place a value in a field from left to right;
  if the data doesn't fill the field, blanks are added
  on the right; if the data is too long, the extra characters
  on the right are ignored. LSET is a left-justify function."
```

Line 120 left-justifies the value in quotes into the first field in buffer 1. Line 130 does the same thing to its quoted string.

Note: RSET would place filler-blanks to the **left** of the item. Truncation would still be on the right.

Now that the data is in the buffer, we can write it to disk with a simple PUT statement:

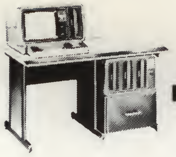
```
  140 PUT 1,1
  150 CLOSE
```

This writes the first record into the file GLOSSARY/BAS.

To read and print the first record in GLOSSARY/BAS, use the following sequence:

```
  160 OPEN"D", 1, "GLOSSARY/BAS"
  170 FIELD 1, 16 AS WD$, 240 AS MEANING$
  180 GET 1,1
  190 PRINT WD$: PRINT MEANING$
  200 CLOSE
```

Lines 160 and 170 are required only because we closed the file in line 150. If we hadn't closed it, we could go directly to line 180.



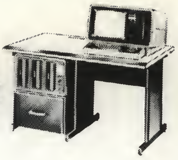
Direct Access: A General Procedure

The above example shows the necessary sequences to read and write using direct access. But it does not demonstrate the primary advantages of this form of access–in particular, it doesn't show how to update existing files by going directly to the desired record.

The program below, GLOSSACC/BAS, develops the glossary example to show some of the techniques of direct access for file maintenance. But before looking at the program, study this general procedure for creating and maintaining files via direct access.

Step	See GLOSSACC/BAS, Line Number
1. Open the file	110
2. Field the buffer	120
3. Get the record to be updated	140
4. Display current contents of the record (use CVD, CVI, CVS before displaying numeric data)	145-170
5. LSET and RSET new values into the fields (use MKD$, MKI$, MKS$ with numeric data before setting it into the buffer)	210-230
6. PUT the updated record	240
7. To update another record, continue at step 3. Otherwise, go to step 8.	250-260
8. Close the file	270

```
100 CLS: CLEAR 300
110 OPEN"D", 1, "GLOSSARY/DAT"
120 FIELD 1, 16 AS WD$, 238 AS MEANING$, 2 AS NX$
130 PRINT"WHAT RECORD TO YOU WANT TO ACCESS";
133 INPUT R%: IF R% <1 THEN PRINT"INVALID RECORD NUMBER": GOTO 130
140 IF R%>LOF(1) THEN 1000
142 GET 1, R%
143 IF ASC(WD$) > 127 THEN PRINT "ENTRY DOESN'T EXIST YET": GOTO 1040
145 NX%=CVI(NX$)          'nx% is the next alphabetical entry
150 PRINT "WORD:  "WD$
160 PRINT "DEF'N": PRINT MEANING$
170 PRINT "NEXT ALPHABETICAL ENTRY IS RECORD #"NX%
180 W$ = "": PRINT: PRINT "TYPE NEW WORD AND PRESS <ENTER> "
```



```
182 INPUT "OR JUST PRESS <ENTER> TO LEAVE UNCHANGED"; W$
190 D$="": PRINT: PRINT "TYPE NEW DEF'N AND PRESS <ENTER>"
192 LINE INPUT "OR JUST PRESS <ENTER> TO LEAVE UNCHANGED? "; D$
200 PRINT: PRINT "CHANGE POINTER TO NEXT ALPHA ENTRY AND PRESS <ENTER>"
202 INPUT "OR JUST PRESS <ENTER> TO LEAVE UNCHANGED"; NX%
210 IF W$<>"" THEN LSET WD$=W$
220 IF D$<>"" THEN LSET MEANING$=D$
230 LSET NX$=MKI$(NX%)
240 PUT 1,R%
245 R%=NX%       'use pointer as default for next record
250 CLS: PRINT "PRESS <ENTER> TO READ THE NEXT ALPHA ENTRY":
PRINT"TYPE RECORD NUMBER <ENTER> FOR ANY OTHER ENTRY":
PRINT"OR TYPE 0 <ENTER> TO QUIT";
255 INPUT R%
260 IF 0<R% THEN 140
270 CLOSE
280 END
1000 PRINT "RECORD NUMBER EXCEEDS END OF FILE. EXTEND FILE? (Y/N)";
1010 R$ = INPUT$(1): PRINT
1020 IF R$ = "N" THEN 130
1030 IF R$ <> "Y" THEN PRINT "EXTEND FILE? TYPE Y OR N";: GOTO 1010
1040 LSET WD$ = "": LSET MEANING$ = "": NX% = 0
1050 PUT 1, R%: GOTO 150
```

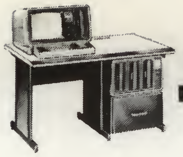
Notice we've added a field, NX$, to the record (line 120). NX$ will contain the number of the record which comes next in alphabetical sequence. This enables us to proceed alphabetically through the glossary, provided we know which record contains the entry which should come first.

For example, suppose the glossary contains:

record#	word (WD$)	defn,	pointer to next alpha. entry (NX$)
1	LEFT-JUSTIFY	...	3
2	BYTE	...	4
3	RIGHT-JUSTIFY	...	0
4	HEXADECIMAL	...	1

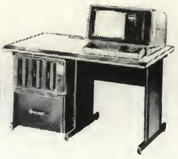
When we read record 2 (BYTE), it tells us that record 4 (HEXADECIMAL) is next, which then tells us record 1 (LEFT-JUSTIFY) is next, etc. The last entry, record 3 (RIGHT-JUSTIFY), points us to zero, which we take to mean "The End".

Sinced NX$ will contain an integer, we have to first convert that number to a two-byte string representation, using MKI$ (line 230 above).



The following program displays the glossary in alphabetical sequence:

```
300 '             *** GLOSSOUT/BAS ***
310 CLS: CLEAR 300
320 OPEN"D", 1, "GLOSSARY/DAT"
330 FIELD 1, 16 AS WD$, 238 AS MEANING$, 2 AS NX$
340 INPUT"WHICH RECORD IS FIRST ALPHABETICALLY"; NX%
350 GET 1, NX%
360 PRINT: PRINT WD$
370 PRINT MEANING$
380 NX%=CVI(NX$)         'get next record number
390 INPUT"PRESS <ENTER> TO CONTINUE";X
400 IF NX%<>0 THEN 350
410 CLOSE
420 END
```



Overlapping Fields

Suppose you want to access a field in two ways—in total and in part. Then you can assign two field names to the same area of the buffer.

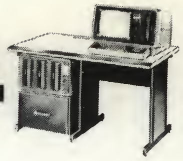
For example, if the first two digits of a six-digit stock-number specify a category, you might use the following field structure:

```
FIELD 1, 6 AS STOCK$, . . . . . . . . .
FIELD 1, 2 AS CTG$, . . . . . . . . . . . .
```

Now STOCK$ will reference the entire stock-number field, while CTG$ will reference only the first two digits of the number.

Chapter 5

Using the Line Editor



Using the Line Editor

The Line Editor is a powerful set of subcommands which simplifies programming. When you are inputting long application programs, the Editor is a fast and efficient way to debug the program and get it running. There are two ways to activate the Editor:

F1

If you type in a long program line or input to a program, and realize you have made a mistake, you can activate the Editor by hitting the **F1** key before you press **ENTER** . This will activate the Editor and all of its subcommands listed below.

EDIT *line number*

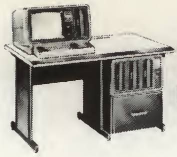
This command starts the Editor when you want to edit program lines which have already been entered. You must specify which line you wish to edit, in one of two ways:

EDIT *line-number* **ENTER**	Lets you edit the specified line. If line number is not in use, an FC error occurs
or	
EDIT.	Lets you edit the current program line – last line entered or altered or in which an error has occurred.

For example, type in and **ENTER** the following line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I^ 2, I ^3 : NEXT
```

This line will be used in exercising all the Edit subcommands described below.



Now type EDIT 100 and press **ENTER** . The Computer will display:

```
100■
```

This starts the Editor. You may begin editing line 100.

Note: EDITing a program line automatically clears all variable values and eliminates pending FOR/NEXT and GOSUB operations. If BASIC encounters a syntax error during program execution, it will automatically put you in the EDIT mode. Before EDITing the line you may want to examine current variable values. In this case, you must type Q as your first EDIT command. This will return you to the command mode, where you may examine variable values. Any other EDIT command (typing E, pressing **ENTER** , etc.) will clear out all variables.

ENTER key

Hitting **ENTER** while in the Edit Mode causes the Computer to record all the changes you've made (if any) in the current line, and returns you to the Command Mode.

*n*Space-bar

In the Edit Mode, hitting the Space-Bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the Computer in the Edit Mode so the Display shows:

```
100■
```

Now hit the Space-Bar. The cursor will move over one space, and the first character of the program line will be displayed. If this character was a blank, then a blank will be displayed. Hit the Space-Bar until you reach the first non-blank character:

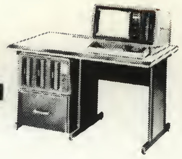
```
100 F ■
```

is displayed. To move over more than one space at a time, hit the desired number of spaces first, and then hit the Space-Bar. For example, type 5 and hit Space-Bar, and the display will show something like this (may vary depending on how many blanks you inserted in the line):

```
100 FOR I= ■
```

Now type 8 and hit the Space-Bar. The cursor will move over 8 spaces to the right, and 8 more characters will be displayed.

```
100 FOR I = 1 TO 10 ■
```



ESC

Hitting the ESC key effects an escape from any of the Insert subcommands listed below: X, I and H. After escaping from an Insert subcommand, you'll still be in the Edit Mode, and the cursor will remain in its current position.. (Hitting ENTER is another way to exit these Insert subcommands).

L (List Line)

When the Computer is in the Edit Mode, and is not currently executing one of the subcommands below, hitting L causes the remainder of the program line to be displayed. The cursor drops down to the next line of the Display, reprints the current line number, and moves to the first position of the line. For example, when the Display shows

```
100 ■
```

hit L (without hitting ENTER key) and line 100 will be displayed:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I∧2, I∧3 : NEXT
100 ■
```

This lets you look at the line in its current form while you're doing the editing.

X (Extend Line)

Causes the rest of the current line to be displayed, moves cursor to end of line, and puts Computer in the Insert subcommand mode so you can add material to the end of the line. For example, using line 100, when the Display shows

```
100 ■
```

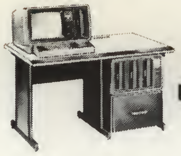
hit X (without hitting ENTER) and the entire line will be displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I∧2, I∧3 :NEXT ■
```

We can now add another statement to the line, or delete material from the line by using the BACKSPACE key. For example, type :PRINT"DONE" at the end of the line. Now hit ENTER . If you now type LIST 100, the Display should show something like this:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I∧2, I∧3 : NEXT : PRINT"DONE" ■
```

Note: If you want to continue editing the line, press ESC to get out of the "X" command mode.



I (Insert)

Allows you to insert material beginning at the current cursor position on the line. (Hitting BACKSPACE will actually delete material from the line in this mode.) For example, type and ENTER the EDIT 100 command, then use the Space Bar to move over to the decimal point in line 100. The Display will show:

```
100 FOR I=1 TO 10 STEP .■
```

Suppose you want to change the increment from .5 to .25. Hit the I key (don't hit ENTER) and the Computer will now let you insert material at the current position. Now hit 2 so the Display shows:

```
100 FOR I=1 TO 10 STEP .2■
```

You've made the necessary change, so press ESC to escape from the Insert Subcommand. Now press the L key to display the remainder of the line and move the cursor back to the beginning of the line:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I∧2, I∧3 : NEXT : PRINT "DONE"
100■
```

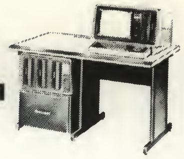
You can also exit the Insert subcommand and save all changes by pressing ENTER . This will return you to Command mode.

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes already made. For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first press ESC (to escape from any subcommand you may be executing); then hit A. (The cursor will drop down to the next line, display the line number and move to the first program character.

E (Save Changes and Exit)

Causes Computer to end editing and save all changes made. You must be in Edit Mode, not executing any subcommand, when you press E to end editing.



Q (Cancel and Exit)

Tells Computer to end editing and cancel all changes made in the current editing session. If you've decided not the change the line, type Q to cancel changes and leave Edit Mode.

If a syntax errors is detected during program execution, BASIC will start the Editor. To examine variable values, you must press Q before typing any other command.

H (Hack and Insert)

Tells Computer to delete remainder of line and lets you insert material at the current cursor position. Hitting BACKSPACE will actually delete a character from the line in this mode. For example, using line 100 listed above, enter the Edit Mode and space over to the last statement, PRINT"DONE". Suppose you wish to delete this statement and insert an END statement. Display will show:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I^2, I^3 : NEXT : ■
```

Now type H and then type END. Press ENTER . List the line:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I^2, I^3 : NEXT : END
```

should be displayed.

Note: To continue editing the line, type the ESC key to get you out of the "H" subcommand.

*n*D (Delete)

Tells Computer to delete the specified number *n* characters to the right of the cursor. The deleted characters will enclosed in backslashes to show you which characters were affected. For example, using line 100, space over to the PRINT command statement:

```
100 FOR I=1 TO 10 STEP .25 : ■
```

Now type 19D. This tells the Computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I=1 TO 10 STEP .25 :  PRINT I, I^2, I^3 : \■
```

When you list the complete line, you'll see that the PRINT statement has been deleted.

*n*C (Change)

Tells the Computer to let you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the Computer assumes you want to change one character. When you have entered *n* number of characters, the Computer returns you to the Edit Mode (so you're not in the *n*C Subcommand). For example, using line 100, suppose you want to change the final value of the FOR-NEXT loop, from "10" to "15". In the Edit Mode, space over to just before the "0" in "10".

```
100 FOR I=1 TO 1 ■
```

Now type C. Computer will assume you want to change just one character. Type 5, then hit L. When you list the line, you'll see that the change has been made.

```
100 FOR 1=1 TO 15 STEP .25 : NEXT : END ■
```

would be the current line if you've followed the editing sequence in this chapter.

*n*S*c* (Search)

Tells the Computer to search for the *n*th occurrence of the character *c,* and move the cursor to that position. If you don't specify a value for *n,* the Computer will search for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line. Note: The Computer only searches through characters to the right of the cursor.

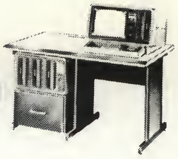
For example, using the current form of line 100, type EDIT 100 ENTER and then hit 2S:. This tells the Computer to search for the second occurence of the colon character. Display should show:

```
100 FOR I=1 TO 15 STEP .25 : NEXT ■
```

n BACKSPACE

Moves the cursor to the left by *n* spaces. If no number *n* is given, the cursor moves back one space. When the cursor backspaces, all characters in its path are erased from the display, but they are not deleted from the program. Use the SPACEBAR to advance the cursor forward and re-display the erased characters.

Note: In any of the insert modes (I, H, and X), BACKSPACE *does* delete characters from the program line.



You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the Insert subcommand, then type the variable name, I. That's all you want to insert, so hit ESC to escape from the Insert subcommand. The next time you list the line, it should appear as:

```
100 FOR I=1 TO 15 STEP .25 : NEXT I: END
```

*n*K*c* (Search and "Kill")

Tells the Computer to delete all characters up to the *n*th occurrence of character *c*, and move the cursor to that position. For example, using the current version of line 100, suppose we want to delete the entire line up to the END statements. Type EDIT 100 (ENTER), and then type 2K: This tells the Computer to delete all characters up to the 2nd occurrence of the colon.

```
100\FOR I=1 TO 15 STEP .25 : NEXT I \■
```

The second colon still needs to be deleted, so type D. The Display will now show:

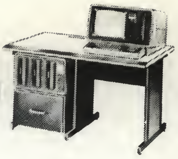
```
100 \ FOR I=1 TO 15 STEP .25 : NEXT I \ \ : \ ■
```

Now hit ENTER and type LIST 100 ENTER

Line 100 should look something like this:

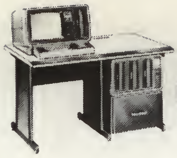
```
100 END
```

Appendix

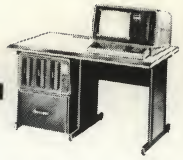


A/Error Messages

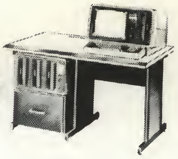
Code	Abbreviation	Explanation
1	NF	**NEXT without FOR.** NEXT is used without a matching FOR statement. This error may also occur if NEXT variables are reversed in a nested loop.
2	SN	**Syntax.** This is usually the result of incorrect punctuation, an illegal character or a misspelled command.
3	RG	**RETURN without GOSUB.** A RETURN statement was encountered before a matching GOSUB was executed.
4	OD	**Out of data.** A READ statement was executed with insufficient data available. The DATA statement may have been left out or all data may have been read.
5	FC	**Illegal function call.** An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative array dimension, negative or zero LOG arguments.
6	OV	**Overflow.** The magnitude of the number derived or input is too large for the data storage type assigned to it. The integer range is [−32768,32767]; other numbers can be in the range $[-1x10^{+38}, -1\times10^{-38}]$ or $[+1\times10^{-38}, +1\times10^{+38}]$. **Note:** There is no underflow error; numbers smaller than +/−1.701411E−38 (single-precision) or +/−1.701411834544556E−38 (double-precision) are rounded to 0.
7	OM	**Out of memory.** All available memory has been used or reserved. This may occur with large array dimensions and nested branches such as GOSUB and FOR/NEXT loops.
8	UL	**Undefined line.** An attempt was made to reference a nonexistent line.
9	BS	**Bad subscript.** An attempt was made to assign an array element with a subscript beyond the dimensioned range.
10	DD	**Double-dimensioned array.** An attempt was made to Dimension an array which had previously been created with DIM or by default statements. ERASE must be used first.
11	/0	**Division by zero.** An attempt was made to use a value of zero in the denominator. **Note:** If you can't find an obvious division by zero, check for division by numbers smaller than allowable ranges (see OV above).
12	ID	**Illegal direct.** An attempt was made to use a program-only statement like INPUT in an immediate (non-program) line.
13	TM	**Type mismatch.** An attempt was made to assign a number to a string variable or a string to a numeric variable.



Code	Abbre-viation	Explanation
14	OS	**Out of string space.** The amount of string space allocated was exceeded. Use CLEAR to allocate more string space. 100 bytes is the default string space allocation.
15	LS	**Long string.** A string variable was assigned a string which exceeded 255 characters in length.
16	ST	**String too complex.** A string operation was too complex to handle. The operation must be broken into shorter steps.
17	CN	**Can't continue.** A CONT command was given at a point where the command can't be carried out, e.g. directly after the program has been edited.
18	UF	**Undefined user function.** An attempt has been made to call a USR function without first defining its entry point via a DEFUSR statement.
19	NR	**No RESUME.** During an error-trapping routine, BASIC has reached the end of the program without encountering a RESUME.
20	RW	**RESUME without error.** A RESUME was encountered when no error was present. You need to insert END or GOTO in front of the error-handling routine.
21	UE	**Undefined error.** Reserved for future use.
22	MO	**Missing operand.** An operation was attempted without providing one of the required operands.
23	BO	**Buffer overflow.** An attempt was made to input a data line which has too many characters to be held in the line buffer.
24-49	UE	**Undefined error.** Reserved for future use.
50	FO	**Field overflow.** An attempt was made to Field more characters than the direct-access file record length allows. The record length is assigned when the file is first Opened. The default length is 256.
51	IE	**Internal error.** Also indicates an attempt to use EOF on a file which is not Open.
52	BN	**Bad file number.** An attempt was made to use a file number which specifies a file that is not Open or that is greater than the number of files specified when BASIC was started up.
53	FF	**File not found.** Reference was made in a LOAD, KILL or OPEN statement to a file which did not exist on the diskette specified.
54	BM	**Bad file mode.** Program attempted to perform direct access on a file Opened for sequential access or vice-versa.



Code	Abbre-viation	Explanation
55	AO	**File already Open.** An attempt was made to Open a file that was already open. This error is also output if a KILL statement is given for an Open file.
56	IO	**Disk I/O error.** An error has been detected during a disk read.
57	FE	**Undefined** in Model II BASIC.
58	UE	**Undefined error.** Reserved for future use.
59	DF	**Disk full.** All storage space on the disk has been used. Kill un-needed files or use a formatted, non-full diskette.
60	EF	**End of file.** An attempt was made to read past the end of file.
61	RN	**Bad record number.** In a PUT or GET statement, the record number is either greater than the allowable maximum or equal to zero.
62	NM	**Undefined** in Model II BASIC.
63	MM	**Mode mismatch.** A sequential OPEN was exected for a file that already existed on the disk as a direct access file, or vice versa.
64	UE	**Undefined error.** Reserved for future use.
65	DS	**Direct statement.** A direct statement was encountered during a load of a program in ASCII format. The load is terminated.
66	FL	**Undefined** in Model II BASIC.

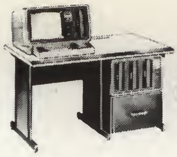


B/Character Codes

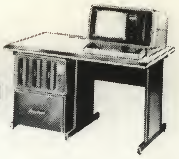
Note: Codes 32-127 represent the ordinary alphanumeric characters. Codes 160-239 represent the graphics-mode alphanumerics, which are used in special applications. For further information, see Chapter 1, "Video Display Output."

Code		Character	
Dec.	Hex.	Keyboard	Video Display
00	00	HOLD	
01	01	F1 CTRL A	Turns on blinking cursor
02	02	F2 CTRL B	Turns off cursor
03	03	BREAK CTRL C	
04	04	CTRL D	Turns on steady cursor
05	05	CTRL E	
06	06	CTRL F	
07	07	CTRL G	
08	08	BACKSPACE CTRL H	Backspaces cursor and erases character
09	09	TAB CTRL I	Advances cursor to next 8-character boundary
10	0A	CTRL J	Line feed
11	0B	CTRL K	
12	0C	CTRL L	
13	0D	ENTER CTRL M	Carriage return
14	0E	CTRL N	
15	0F	CTRL O	
16	10	CTRL P	
17	11	CTRL Q	
18	12	CTRL R	
19	13	CTRL S	
20	14	CTRL T	
21	15	CTRL U	
22	16	CTRL V	
23	17	CTRL W	Erases to end of line
24	18	CTRL X	Erases to end of screen
25	19	CTRL Y	Sets white-on-black mode
26	1A	CTRL Z	Sets black-on-white mode
27	1B	ESC	Clears screen, homes cursor

*BREAK is always intercepted. It will never return a code 3 to the user program.

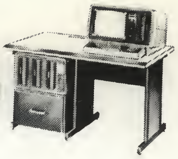


Code		Character	
Dec.	Hex.	Keyboard	Video Display
28	1C	←	Moves cursor back
29	1D	→	Moves cursor forward
30	1E	↑	Sets 80-character mode and clears Display
31	1F	↓	Sets 40-character mode and clears Display
32	20	SPACE BAR	b̸
33	21	!	!
34	22	"	"
35	23	#	#
36	24	$	$
37	25	%	%
38	26	&	&
39	27	'	'
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	–	–
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	@	@
65	41	A	A
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G



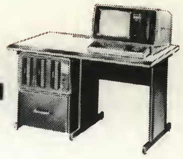
Code		Character	
Dec.	Hex.	Keyboard	Video Display
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	[	[
92	5C	CTRL-9	\
93	5D	]	]
94	5E	^	^
95	5F	_	_
96	60		`
97	61	A	a
98	62	B	b
99	63	C	c
100	64	D	d
101	65	E	e
102	66	F	f
103	67	G	g
104	68	H	h
105	69	I	i
106	6A	J	j
107	6B	K	k
108	6C	L	l
109	6D	M	m
110	6E	N	n
111	6F	O	o
112	70	P	p
113	71	Q	q
114	72	R	r
115	73	S	s
116	74	T	t

Code		Character	
Dec.	Hex.	Keyboard	Video Display
117	75	U	u
118	76	V	v
119	77	W	w
120	78	X	x
121	79	Y	y
122	7A	Z	z
123	7B	{	{
124	7C	CTRL-0	¦
125	7D	}	}
126	7E	CTRL-6	~
127	7F		±
128	80		▙
129	81		┓
130	82		┛
131	83		▌
132	84		┳
133	85		┫
134	86		┻
135	87		┣
136	88		┰
137	89		┨
138	8A		┸
139	8B		┠
140	8C		╋
141	8D		╉
142	8E		┼
143	8F		┽
144	90		▁
145	91		▂
146	92		▃
147	93		▄
148	94		▌
149	95		▏
150	96		━
151	97		─
152	98		▁
153	99		▂
154	9A		▅
155	9B		▆
156	9C		▌
157	9D		▐
158	9E		█
159	9F		↑
160	A0		b̸
161	A1		!
162	A2		"



Code		Character	
Dec.	Hex.	Keyboard	Video Display
163	A3		#
164	A4		$
165	A5		%
166	A6		&
167	A7		'
168	A8		(
169	A9		)
170	AA		*
171	AB		+
172	AC		,
173	AD		–
174	AE		.
175	AF		/
176	B0		0
177	B1		1
178	B2		2
179	B3		3
180	B4		4
181	B5		5
182	B6		6
183	B7		7
184	B8		8
185	B9		9
186	BA		:
187	BB		;
188	BC		{
189	BD		=
190	BE		}
191	BF		?
192	C0		@
193	C1		A
194	C2		B
195	C3		C
196	C4		D
197	C5		E
198	C6		F
199	C7		G
200	C8		H
201	C9		I
202	CA		J
203	CB		K
204	CC		L
205	CD		M
206	CE		N
207	CF		O
208	D0		P

Code		Character	
Dec.	Hex.	Keyboard	Video Display
209	D1		Q
210	D2		R
211	D3		S
212	D4		T
213	D5		U
214	D6		V
215	D7		W
216	D8		X
217	D9		Y
218	DA		Z
219	DB		[
220	DC		\
221	DD		]
222	DE		^
223	DF		_
224	E0		`
225	E1		a
226	E2		b
227	E3		c
228	E4		d
229	E5		e
230	E6		f
231	E7		g
232	E8		h
233	E9		i
234	EA		j
235	EB		k
236	EC		l
237	ED		m
238	EE		n
239	EF		o
240	F0	Unused	
241	F1	Unused	
242	F2	Unused	
243	F3	Unused	
244	F4	Unused	
245	F5	Unused	
246	F6	Unused	
247	F7	Unused	
248	F8	Unused	
249	F9	Unused	
250	FA	Unused	
251	FB	Unused	
252	FC		Moves cursor left
253	FD		Moves cursor right
254	FE		Moves cursor up
255	FF		Moves cursor down

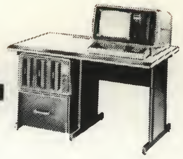


C/Reserved Words

A reserved word with a dollar-sign ("$") after it may be used as a numeric variable name if the dollar-sign is dropped. For instance, CHR and CHR# are valid variable names. However, DEF statements may not be used to assign values to this type of variable.

ABS
AND
ASC
ATN
AUTO
CDBL
CHR$
CINT
CLEAR
CLOCK
CLOSE
CLS
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFSTR
DEFUSR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FORMAT
FRE
FREE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INPUT
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LINEINPUT
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LPRINT
LSET
MEM
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OCT$
ON
OPEN
OR
POINT
POS
POSN
PRINT
PUT
RANDOM
READ
REM
RENAME
RENUM
RESTORE
RESUME
RETURN
RIGHT$
RND
ROW
RSET
RUN
SAVE
SGN
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VERIFY

XOR

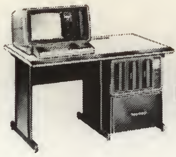


Internal Codes for BASIC Keywords

To save space, BASIC keywords are stored in memory and in compressed (non-ASCII) disk files as one-byte codes. To determine the sequence for a multi-word keyword sequence, simply string together the codes of the individual keywords. For example, ON ERROR GOTO is stored as 159 157 138.

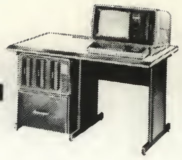
Keyword	Code	
	Dec	Hex
ABS	219	DB
AND	207	CF
ASC	249	F9
ATN	230	E6
AUTO	180	B4
CDBL	242	F2
CHR$	250	FA
CINT	240	F0
CLEAR	181	B5
CLOCK	67	43
CLOSE	164	A4
CLS	130	82
CONT	176	B0
COS	227	E3
CSNG	241	F1
CVD	233	E9
CVI	231	E7
CVS	232	E8
DATA	133	85
DATE$	196	C4
DEFDBL	154	9A
DEF	174	AE
DEFINT	152	98
DEFSNG	153	99
DEFSTR	151	97
DELETE	179	B3
DIM	135	87
EDIT	156	9C
END	128	80
EOT	234	EA
EQV	210	D2
ERASE	150	96
ERL	191	BF
ERR	192	C0
ERROR	157	9D
EXP	226	E2
FIELD	161	A1
FILES	70	46
FIX	243	F3
FN	186	BA
FOR	129	81
FRE	220	DC
GET	162	A2
GOSUB	142	8E
GOTO	138	8A
HEX$	246	F6
IF	140	8C
IMP	211	D3
INKEY$	198	C6
INPUT	134	86
INPUT$*	134	86
INSTR	194	C2
INT	218	DA
KILL	168	A8
LEFT$	252	FC
LEN	244	F4
LET	137	89
LINE	155	9B
LIST	177	B1
LLIST	178	B2
LOAD	165	A5
LOC	235	EB
LOF	236	EC
LOG	225	E1
LPRINT	173	AD
LSET	169	A9
MEM	197	C5
MERGE	166	A6
MID$	254	FE
MKD$	239	EF
MKI$	237	ED
MKS$	238	EE

*The dollar-sign $ is stored as an ASCII character (36) following the code for INPUT.



Keyword	Code Dec	Code Hex
MOD	212	D4
NEW	183	B7
NEXT	132	84
NOT	200	C8
OCT$	245	F5
ON	159	9F
OPEN	160	A0
OR	208	D0
POS	222	DE
PRINT	175	AF
PUT	163	A3
RANDOM	131	83
READ	136	88
REM	144	90
RENUM	182	B6
RESTORE	141	8D
RESUME	158	9E
RETURN	143	8F
RIGHT$	253	FD
RND	224	E0
ROW	221	DD
RSET	170	AA
RUN	139	8B
SAVE	171	AB
SGN	217	D9

Keyword	Code Dec	Code Hex
SIN	228	E4
SPACE$	251	FB
SPC	83	53
SQR	223	DF
STEP	201	C9
STOP	145	91
STR$	247	F7
STRING$	193	C1
SWAP	149	95
SYSTEM	172	AC
TAB	84	54
TAN	229	E5
THEN	99	C7
TIME$	195	C3
TO	185	B9
TROFF	148	94
TRON	147	93
USING	188	BC
USR	190	BE
VAL	248	F8
VARPTR	189	BD
VERIFY	86	56
XOR	209	D1



E/Glossary

access The method in which information is read from or written to disk; see **direct access** and **sequential access.**

address A location in memory, usually specified as a two-byte hexadecimal number. The address range [0 to FFFF] is represented in decimal as [0 to 32767] [−32768, . . ., −1].

alphabetic Referring strictly to the letters A to Z.

alphanumeric Referring to the set of letters A to Z and the numerals 0-9.

argument The string or numeric quantity which is supplied to a function and is then operated on to derive a result; this result is referred to as the **value** of the function.

array An organized set of elements which can be referenced in total or individually, using the array name and one or more subscripts. In BASIC, any variable name can be used to name an array; and arrays can have one or more dimensions. AR( ) signifies a one-dimensional array named AR; AR(,) signifies a two-dimensional array named AR; etc.

ASCII American Standard Code for Information Interchange. This method of coding is used to store textual data. Numeric data is typically stored in a more compressed format.

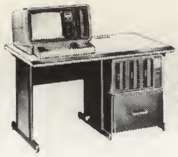
ASCII format disk file Disk files in which each byte corresponds to one character of the original data. For example, a BASIC program stored in ASCII format "looks like" the program listing, with each character ASCII encoded. Compare to **compressed-format** file.

backup diskette An exact copy of the original: a "safe copy". You should keep backups of your original TRSDOS diskette and all important data diskettes.

BASIC Beginners' All-purpose Symbolic Instruction Code.

binary Having two possible states, e.g., the binary digits 0 and 1. The binary (base 2) numbering system uses sequences of zeroes and ones to represent quantities. This is analagous to the Computer's internal representation of data, using electrical values for 0 and 1.

bit Binary digit; the smallest unit of memory in the Computer, capable of representing the values 0 and 1.



break To interrupt execution of a program. In BASIC the statement STOP causes a break in execution, as does pressing the BREAK key.

buffer An area in RAM where data is accumulated for further processing. For example, to pass data from BASIC to a disk file, and vice-versa, the data must go through a file-buffer.

buffer field A portion of the buffer which you define as the storage area for a buffer-field variable. Dividing a buffer into fields allows you to include multiple values in one logical record.

byte The smallest addressable unit of memory in the Computer, consisting of 8 consecutive bits, and capable of representing 256 different values, e.g., decimal values from 0 to 255.

compressed-format A method of storing information in less space than a standard ASCII representation would require. An integer always requires two bytes; a single-precision number, four; a double-precision number, 8 – regardless of how many characters are required to represent the numbers as text. String values are not stored in compressed format; each character requires one byte.

BASIC programs in RAM and non-ASCII disk files are stored in compressed-format, with all BASIC keywords stored as special one-byte codes.

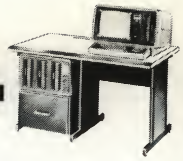
close Terminate access to a disk file. Before re-accessing the file, you must re-open it.

data Information that is passed to or output from a program. There are four types of data:
- Integer numbers
- Single-precision numbers
- Double-precision numbers
- Character-string sequences (strings)

debug To find and remove logical or syntactic errors from a program.

decimal Capable of assuming one of ten states, e.g., the decimal digits 0,1,...,9. Decimal (base 10) numbering is the everyday system, using sequences of decimal digits. Decimal numbers are stored in binary code in Model II BASIC.

default An action or value which is supplied by a program when you do not specify an action or value to be used.



delimiter A character which marks the beginning or end of a data item, and is not a part of the data. For example, the double-quote symbol is a string delimiter to BASIC.

destination The device or address which receives the data during a data transfer operation. For example, during a BACKUP operation, the destination disk is the one onto which the source disk is being copied.

device A physical part of the computer system used for data I/O, e.g., keyboard, display, line printer, disk drive.

directory A listing of the files which are contained on a disk.

direct access A means of processing any record in a file. Contrast with **sequential access.**

diskette A magnetic recording medium for mass data storage.

drive specification An optional field in a TRSDOS file specification and in some TRSDOS commands, consisting of a colon followed by one of the digits 0 through 3. The drive specification is used to specify which drive is to be used for a disk read or write.

When the drive specification is omitted from a command involving a read operation, TRSDOS will search through all the disks for the desired file, starting with drive 0.

When the drive specification is omitted from a command involving a write operation, TRSDOS will generally search through all non write protected drives for the desired file.

drive number An integer value from 0 to 3, specifying one of the disk drives.

dummy variable A variable name which is used in an expression to meet syntactic requirements, but whose value is insignificant.

edit To change existing information.

entry point The address of a machine-language program or routine where execution is to begin. This is not necessarily the same as the starting address. Entry point is also referred to as the **transfer address.**

field A user-defined subdivision of a direct access file-buffer, created and named with the FIELD statement.

field name A string variable which has been assigned to a field in a direct access file-buffer via the FIELD statement.

file An organized collection of related data. Under TRSDOS, a file is the largest block of information on a diskette which can be addressed with a single command. BASIC programs and data are stored on disk in distinct files.

file extension An optional field in a file specification, consisting of a diagonal slash "/" followed by up to three alphanumeric characters; the extension can be used to identify the file type, e.g., /BAS, /TXT, /MIM, for BASIC, text, and memory image, respectively.

file name A required field in a file specification, consisting of one alphabetic character followed by up to 7 alphanumeric characters. File names are assigned when a file is created or renamed.

file specification A sequence of characters which specifies a particular disk file under TRSDOS, consisting of a mandatory file name, followed by an optional extension, password, drive specification and diskette name.

format To organize a new or magnetically erased diskette into tracks and sectors, via the TRSDOS FORMAT utility.

granule The smallest unit of allocatable space on a disk, consisting of 5 sectors.

hexadecimal or **hex** Capable of existing in one of 16 possible states. For example, the hexadecimal digits are 0,1,2,. . .,9,A,B,C,D,E,F. Hexadecimal (base-16) numbers are sequences of hexadecimal digits. Address and byte values are frequently given in hexadecimal form. In Model II BASIC, hexadecimal constants can be input by prefixing the constant with &H.

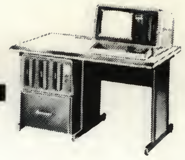
increment The value which is added to a counter each time one cycle of a repetitive procedure is completed.

input To transfer data from outside the Computer (from a disk file, keyboard, etc.) into RAM.

kilobyte or **K** 1024 bytes of memory. Thus a 64K System includes 64*1024=65536 bytes of memory.

logical expression An expression which is evaluated as either True (=−1) or FALSE (=0).

logical record A block of data which can be processed as a unit. In a disk file, the record length is set when the file is first Opened. Sequential access files have a record length of 1; direct access files have a record length in the range [1,256].



machine language The Z-80A instruction set, usually specified in hexadecimal code. All higher-level languages must be translated into machine-language, or interpreted by machine language, in order to be executed by the Computer.

null string A string which has a length of zero. For example, the assignment
A$=""
makes A$ a null string.

object code Machine language derived from "source code", typically, from assembly language.

octal Capable of existing in one of 8 states, for example, the octal digits are 0,1, . . .,7. Octal (base-8) numbers are sequences of octal digits. Address and byte values are frequently given in octal form. Under model II BASIC, an octal constant can be input by prefixing the octal number with the symbol &O.

open To prepare a file for access by assigning a sequential input, sequential output or direct I/O buffer to it.

output To transfer data from inside the Computer's memory to some external area, e.g., a disk file or a line printer.

parameter Information supplied with a command to specify how the command is to operate.

password An optional field in a filespec consisting of up to 8 alphanumeric characters. If a file is created without a password, 8 blanks become the default password. To access a file, you must specify the password in the filespec.

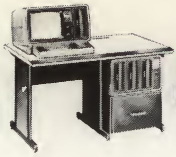
Using the TRSDOS ATTRIB command, you can assign both update and access passwords; the access password will grant only a limited degree of access, while the update password grants total access to the file. See **file specification.**

prompt A character or message provided by a program to indicate that it's ready to accept keyboard input.

protected file A disk file which has a non-blank password, and therefore can only be accessed by reference to that password.

protection level The degree of access granted by using the access password: kill, rename, write, read, or execute.

random access memory or **RAM** Semiconductor memory which can be addressed directly and either read from or written to.



routine A sequence of instructions to carry out a certain function; typically, a routine called from multiple points in a program.

sector A physical record on the diskette, containing 256 bytes of data. The unit of data transferred for the user is called a logical record, and can contain from one to 256 bytes.

sequential access Reading from a disk file or writing to it "from start to finish", without being able to directly access a particular record in the file.

statement A complete instruction in BASIC.

string Any sequence of characters which must be examined verbatim for meaning: in other words, the string does not correspond to a quantity. For example, the *number* 1234 represents the same quantity as 1000+234, but the *string* "1234" does not. (String addition is actually concatenation, or stringing-together, so that: "1234" equals "1" + "2" + "3" + "4").

syntax The "grammatical" requirements for a command or statement. Syntax generally refers to punctuation and ordering of elements within a statement.

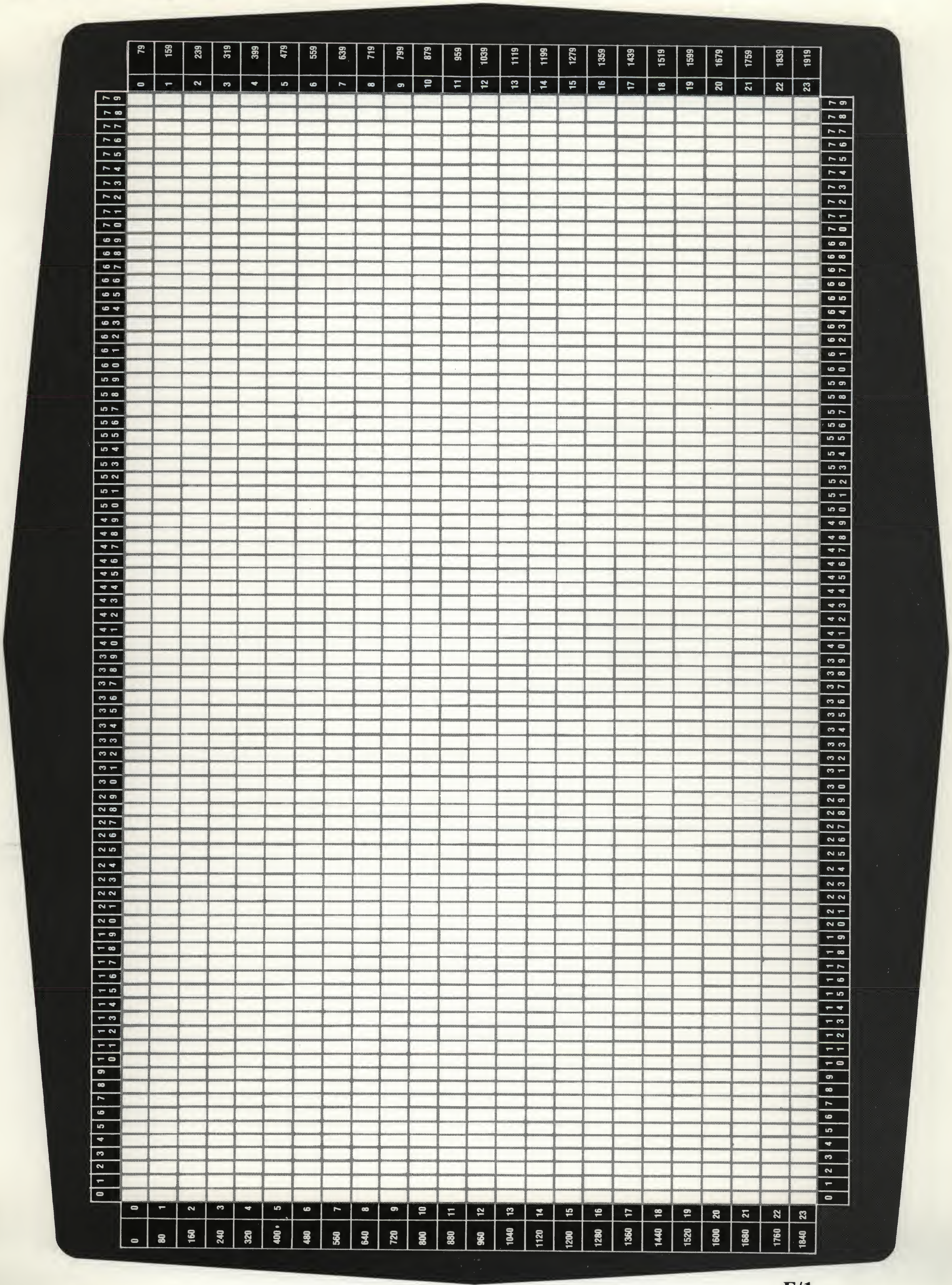
transfer address See **entry point.**

TRSDOS TRS-80 Disk Operating System, pronounced "triss-doss". TRSDOS is supplied on disk and is then loaded into RAM.

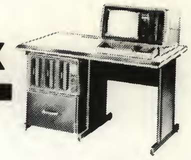
utility A program or routine which serves a limited, specific purpose. There are two extended TRSDOS utilities, FORMAT and BACKUP.

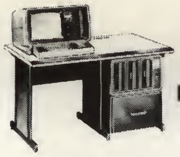
write-protect To physically protect a disk from being written to by leaving the write-protect notch uncovered.

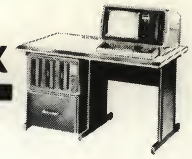
F/Video Display Worksheet



Index







IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

For a period of 90 days from the date of delivery, Radio Shack warrants to the original purchaser that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty is only applicable to purchases from Radio Shack company-owned retail outlets and through duly authorized franchisees and dealers. The warranty shall be void if this unit's case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair, and proof of purchase must be presented. Purchaser's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including, but not limited to, statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

RADIO SHACK A DIVISION OF TANDY CORPORATION
U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.
8749124
8749122
8749123